

UNIX - Command-Line Survival Guide

Files, directories, commands, text editors

Simon Prochnik & Lincoln Stein

Book Chapters

Learning Perl (6th ed.): Chap. 1
Unix & Perl to the Rescue (1st ed.): Chaps. 3 & 5

Lecture Notes

- [What is the Command Line?](#)
 - [Logging In](#)
 - [The Desktop](#)
 - [The Shell](#)
 - [Home Sweet Home](#)
 - [Getting Around](#)
 - [Running Commands](#)
 - [Command Redirection](#)
 - [Pipes](#)
-

What is the Command Line?

Underlying the pretty Mac OSX GUI is a powerful command-line operating system. The command line gives you access to the internals of the OS, and is also a convenient way to write custom software and scripts.

Many bioinformatics tools are written to run on the command line and have no graphical interface. In many cases, a command line tool is more versatile than a graphical tool, because you can easily combine command line tools into automated scripts that accomplish tasks without human intervention.

In this course, we will be writing Perl scripts that are completely command-line based.

Logging into Your Workstation

Your workstation is an iMac. To log into it, provide the following information:

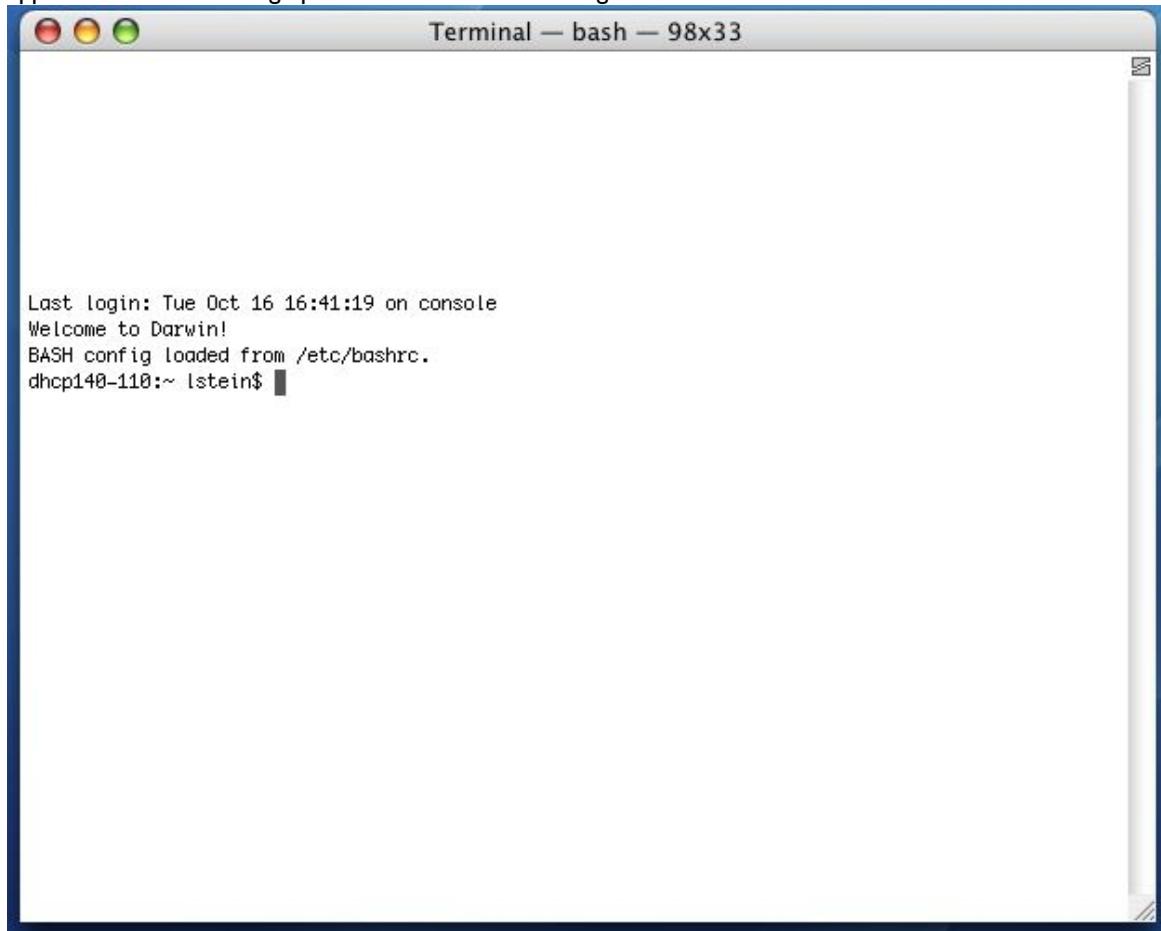
Your username: the initial of your first name, followed by your full last name. For example, if your username is **srobb** for **sofia robb**

Your password: **changeme**

Bringing up the Command Line

To bring up the command line, use the Finder to navigate to *Applications->Utilities* and double-click on the *Terminal*

application. This will bring up a window like the following:



OSX Terminal

You can open several Terminal windows at once. This is often helpful.

You will be using this application a lot, so I suggest that you drag the Terminal icon into the shortcuts bar at the bottom of your screen.

OK. I've Logged in. What Now?

The terminal window is running a **shell** called "bash." The shell is a loop that:

1. Prints a prompt
2. Reads a line of input from the keyboard
3. Parses the line into one or more commands
4. Executes the commands (which usually print some output to the terminal)
5. Go back 1.

There are many different shells with bizarre names like **bash**, **sh**, **csh**, **tcsh**, **ksh**, and **zsh**. The "sh" part means shell. Each shell was designed for the purpose of confusing you and tripping you up. We have set up your accounts to use **bash**. Stay with **bash** and you'll get used to it, eventually.

Command-Line Prompt

Most of bioinformatics is done with command-line software, so you should take some time to learn to use the shell effectively.

This is a command line prompt:

```
bush202>
```

This is another:

```
(~) 51%
```

This is another:

```
srobb@bush202 1:12PM>
```

What you get depends on how the system administrator has customized your login. You can customize yourself when you know how.

The prompt tells you the shell is ready to accept a command. When a long-running command is going, the prompt will not reappear until the system is ready to deal with your next request.

Issuing Commands

Type in a command and press the <Enter> key. If the command has output, it will appear on the screen. Example:

```
(~) 53% ls -F
GNUstep/
INBOX
INBOX~
Mail@
News/
axhome/
bin/
build/
ccod/
(~) 54%
cool_elegans.movies.txt  man/
docs/          mtv/
etc/           nsmail/
games/         pcod/
get_this_book.txt projects/
jcod/          public_html/
lib/           src/
linux/         tmp/
```

The command here is *ls -F*, which produces a listing of files and directories in the current directory (more on which later). After its output, the command prompt appears again.

Some programs will take a long time to run. After you issue their command name, you won't recover the shell prompt until they're done. You can either launch a new shell (from Terminal's File menu), or run the command in the background using the ampersand:

```
(~) 54% long_running_application&
(~) 55%
```

The command will now run in the background until it is finished. If it has any output, the output will be printed to the terminal window. You may wish to redirect the output as described later.

Command Line Editing

Most shells offer command line entering. Up until the comment you press <Enter>, you can go back over the command line and edit it using the keyboard. Here are the most useful keystrokes:

Backspace

Delete the previous character and back up one.

Left arrow, right arrow

Move the text insertion point (cursor) one character to the left or right.

control-a (^a)

Move the cursor to the beginning of the line. Mnemonic: A is first letter of alphabet

control-e (^e)

Move the cursor to the end of the line. Mnemonic: <E> for the End (^Z was already taken for something else).

control-d (^d)

Delete the character currently under the cursor. D=Delete.

control-k (^k)

Delete the entire line from the cursor to the end. k=kill. The line isn't actually deleted, but put into a temporary holding place called the "kill buffer".

control-y (^y)

Paste the contents of the kill buffer onto the command line starting at the cursor. y=yank.

Up arrow, down arrow

Move up and down in the command history. This lets you reissue previous commands, possibly after modifying them.

There are also some useful shell commands you can issue:

history

Show all the commands that you have issued recently, nicely numbered.

!<number>

Reissue an old command, based on its number (which you can get from *history*)

!!

Reissue the immediate previous command.

!<partial command string>

Reissue the previous command that began with the indicated letters. For example !! would reissue the ls -F command from the earlier example.

bash offers automatic command completion and spelling correction. If you type part of a command and then the tab key, it will prompt you with all the possible completions of the command. For example:

```
(~) 51% fd<tab>
(~) 51% fd
fd2ps    fdesign   fdformat fdlist    fdmount   fdmountd fdrawcmd fdumount
(~) 51%
```

If you hit tab after typing a command, but before pressing <Enter>, **bash** will prompt you with a list of file names. This is because many commands operate on files.

Wildcards

You can use wildcards when referring to files. "*" refers to zero or more characters. "?" refers to any single character. For example, to list all files with the extension ".txt", run **ls** with the pattern "*".txt":

```
(~) 56% ls -F *.txt
final_exam_questions.txt  genomics_problem.txt
genebridge.txt            mapping_run.txt
```

There are several more advanced types of wildcard patterns which you can read about in the **tcsesh** manual page. For example, you can refer to files beginning with the characters "f" or "g" and ending with ".txt" this way:

```
(~) 57% ls -F [f-g]*.txt
final_exam_questions.txt  genebridge.txt
                                          genomics_problem.tz
```

Home Sweet Home

When you first log in, you'll be placed in a part of the system that is your personal domain, called the *home directory*. You are free to do with this area what you will: in particular you can create and delete files and other directories. In general, you cannot create files elsewhere in the system.

Your home directory lives somewhere way down deep in the bowels of the system. On our iMacs, it is a directory with the same name as your login name, located in **/Users**. The full directory path is therefore **/Users/username**. Since this is a pain to write, the shell allows you to abbreviate it as `~username` (where "username" is your user name), or simply as `~`. The weird character (technically called the "tilde" or "twiddle") is usually hidden at the upper left corner of your keyboard.

To see what is in your home directory, issue the command `ls -F`:

```
(~) % ls -F
INBOX      Mail/       News/       nsmail/      public_html/
```

This shows one file "INBOX" and four directories ("Mail", "News") and so on. (The "-F" in the command turns on fancy mode, which appends special characters to directory listings to tell you more about what you're seeing. "/" means directory.)

In addition to the files and directories shown with `ls -F`, there may be one or more hidden files. These are files and directories whose names start with a `.` (technically called the "dot" character). To see these hidden files, add an "a" to the options sent to the `ls` command:

```
(~) % ls -af
./          .cshrc        .login        Mail/
../         .fetchhost    .netscape/    News/
.Xauthority .fvwmrc      .xinitrc*   nsmail/
.Xdefaults   .history     .xsession@  public_html/
.bash_profile .less        .xsession-errors
.bashrc      .lessrc      INBOX
```

Whoa! There's a lot of hidden stuff there. But don't go deleting dot files willy-nilly. Many of them are esential configuration files for commands and other programs. For example, the `.profile` file contains configuration information for the **bash** shell. You can peek into it and see all of **bash**'s many options. You can edit it (when you know what you're doing) in order to change things like the command prompt and command search path.

Getting Around

You can move around from directory to directory using the `cd` command. Give the name of the directory you want to move to, or give no name to move back to your home directory. Use the `pwd` command to see where you are (or rely on the prompt, if configured):

```
(~/docs/grad_course/i) 56% cd
(~) 57% cd /
(/) 58% ls -F
bin/      dosc/      gmon.out    mnt/      sbin/
boot/     etc/       home@      net/      tmp/
```

```
cdrom/          fastboot      lib/         proc/        usr/
dev/           floppy/       lost+found/   root/       var/
(/) 59% cd ~/docs/
(~/docs) 60% pwd
/usr/home/lstein/docs
(~/docs) 62% cd ../projects/
(~/projects) 63% ls
Ace-browser/      bass.patch
Ace-perl/         cgi/
Foo/              cgi3/
Interface/        computertalk/
Net-Interface-0.02/ crypt-cbc.patch
Net-Interface-0.02.tar.gz fixer/
Pts/              fixer.tcsh
Pts.bak/          introspect.pl*
PubMed/          introspection.pm
SNPdb/            rhmap/
Tie-DBI/          sbox/
ace/              sbox-1.00/
atir/             sbox-1.00.tgz
bass-1.30a/       zhmapper.tar.gz
bass-1.30a.tar.gz
(~/projects) 64%
```

Each directory contains two special hidden directories named "." and "..". "." refers always to the directory in which it is located. ".." refers always to the parent of the directory. This lets you move upward in the directory hierarchy like this:

```
(~/docs) 64% cd ..
```

and to do arbitrarily weird things like this:

```
(~/docs) 65% cd ../../docs
```

The latter command moves upward to levels, and then into a directory named "docs".

If you get lost, the *pwd* command prints out the full path to the current directory:

```
(~) 56% pwd
/Users/lstein
```

Essential Unix Commands

With the exception of a few commands that are built directly into the shell, all Unix commands are standalone executable programs. When you type the name of a command, the shell will search through all the directories listed in the PATH environment variable for an executable of the same name. If found, the shell will execute the command. Otherwise, it will give a "command not found" error.

Most commands live in /bin, /usr/bin, or /usr/local/bin.

Getting Information About Commands

The **man** command will give a brief synopsis of the command:

```
(~) 76% man wc
Formatting page, please wait...
WC(1)                                     WC(1)
```

NAME

wc - print the number of bytes, words, and lines in files

SYNOPSIS

```
wc [-clw] [--bytes] [--chars] [--lines] [--words] [--help]
[--version] [file...]
```

DESCRIPTION

This manual page documents the GNU version of wc. wc counts the number of bytes, whitespace-separated words,

...

Finding Out What Commands are on Your Computer

The **apropos** command will search for commands matching a keyword or phrase:

```
(~) 100% apropos column
showtable (1)          - Show data in nicely formatted columns
colrm (1)              - remove columns from a file
column (1)             - columnate lists
fix132x43 (1)         - fix problems with certain (132 column) graphics
modes
```

Arguments and Command Switches

Many commands take arguments. Arguments are often (but not inevitably) the names of one or more files to operate on. Most commands also take command-line "switches" or "options" which fine-tune what the command does. Some commands recognize "short switches" that consist of a single character, while others recognize "long switches" consisting of whole words.

The **wc** (word count) program is an example of a command that recognizes both long and short options. You can pass it the **-c**, **-w** and/or **-l** options to count the characters, words and lines in a text file, respectively. Or you can use the longer but more readable, **--chars**, **--words** or **--lines** options. Both these examples count the number of characters and lines in the text file /var/log/messages:

```
(~) 102% wc -c -l /var/log/messages
      23      941 /var/log/messages
(~) 103% wc --chars --lines /var/log/messages
      23      941 /var/log/messages
```

You can cluster short switches by concatenating them together, as shown in this example:

```
(~) 104% wc -cl /var/log/messages
      23      941 /var/log/messages
```

Many commands will give a brief usage summary when you call them with the **-h** or **--help** switch.

Spaces and Funny Characters

The shell uses whitespace (spaces, tabs and other nonprinting characters) to separate arguments. If you want to embed

whitespace in an argument, put single quotes around it. For example:

```
mail -s 'An important message' 'Bob Ghost <bob@ghost.org>'
```

This will send an e-mail to the fictitious person Bob Ghost. The **-s** switch takes an argument, which is the subject line for the e-mail. Because the desired subject contains spaces, it has to have quotes around it. Likewise, my e-mail address, which contains embedded spaces, must also be quoted in this way.

Certain special non-printing characters have escape codes associated with them:

Escape Code	Description
\n	new line character
\t	tab character
\r	carriage return character
\a	bell character (ding! ding!)
\nnn	the character whose ASCII code in octal is nnn

Useful Commands

Here are some commands that are used extremely frequently. Use **man** to learn more about them. Some of these commands may be useful for solving the problem set ;-)

Manipulating Directories

ls

Directory listing. Most frequently used as **ls -F** (decorated listing) and **ls -l** (long listing).

mv

Rename or move a file or directory.

cp

Copy a file.

rm

Remove (delete) a file.

mkdir

Make a directory

rmdir

Remove a directory

ln

Create a symbolic or hard link.

chmod

Change the permissions of a file or directory.

Manipulating Files

cat

Concatenate program. Can be used to concatenate multiple files together into a single file, or, much more frequently, to send the contents of a file to the terminal for viewing.

more

Scroll through a file page by page. Very useful when viewing large files. Works even with files that are too big to be opened by a text editor.

less

A version of **more** with more features.

head	View the head (top) of a file. You can control how many lines to view.
tail	View the tail (bottom) of a file. You can control how many lines to view. You can also use tail to view a growing file.
wc	Count words, lines and/or characters in one or more files.
tr	Substitute one character for another. Also useful for deleting characters.
sort	Sort the lines in a file alphabetically or numerically.
uniq	Remove duplicated lines in a file.
cut	Remove sections from each line of a file or files.
fold	Wrap each input line to fit in a specified width.
grep	Filter a file for lines matching a specified pattern. Can also be reversed to print out lines that don't match the specified pattern.
gzip (gunzip)	Compress (uncompress) a file.
tar	Archive or unarchive an entire directory into a single file.
emacs	Run the Emacs text editor (good for experts).

Networking

ssh	A secure (encrypted) way to log into machines.
ping	See if a remote host is up.
ftp and the secure version sftp	Transfer files using the File Transfer Protocol.
who	See who else is logged in.
lp	Send a file or set of files to a printer.

Standard I/O and Command Redirection

Unix commands communicate via the command line interface. They can print information out to the terminal for you to see, and accept input from the keyboard (that is, from *you!*)

Every Unix program starts out with three connections to the outside world. These connections are called "streams" because they act like a stream of information (metaphorically speaking):

standard input

This is a communications stream initially attached to the keyboard. When the program reads from standard input, it reads whatever text you type in.

standard output

This stream is initially attached to the command window. Anything the program prints to this channel appears in your

terminal window.
standard error

This stream is also initially attached to the command window. It is a separate channel intended for printing error messages.

The word "initially" might lead you to think that standard input, output and error can somehow be detached from their starting places and reattached somewhere else. And you'd be right. You can attach one or more of these three streams to a file, a device, or even to another program. This sounds esoteric, but it is actually very useful.

A Simple Example

The **wc** program counts lines, characters and words in data sent to its standard input. You can use it interactively like this:

```
(~) 62% wc
Mary had a little lamb,
little lamb,
little lamb.

Mary had a little lamb,
whose fleece was white as snow.
^D
6       20      107
```

In this example, I ran the **wc** program. It waited for me to type in a little poem. When I was done, I typed the END-OF-FILE character, control-D (^D for short). **wc** then printed out three numbers indicating the number of lines, words and characters in the input.

More often, you'll want to count the number of lines in a big file; say a file filled with DNA sequences. You can do this by redirecting **wc**'s standard input from a file. This uses the < metacharacter:

```
(~) 63% wc <big_file.fasta
2943     2998     419272
```

If you wanted to record these counts for posterity, you could redirect standard output as well using the > metacharacter:

```
(~) 64% wc <big_file.fasta >count.txt
```

Now if you **cat** the file *count.txt*, you'll see that the data has been recorded. **cat** works by taking its standard input and copying it to standard output. We redirect standard input from the *count.txt* file, and leave standard output at its default, attached to the terminal:

```
(~) 65% cat <count.txt
2943     2998     419272
```

Redirection Meta-Characters

Here's the complete list of redirection commands for **bash**:

< <i>filename</i>	Redirect standard input to file
> <i>filename</i>	Redirect standard output to file
1> <i>filename</i>	Redirect just standard output to file (same as above)
2> <i>filename</i>	Redirect just standard error to file

```
>filename 2>&1 Redirect both stdout and stderr to file
```

These can be combined. For example, this command redirects standard input from the file named `/etc/passwd`, writes its results into the file `search.out`, and writes its error messages (if any) into a file named `search.err`. What does it do? It searches the password file for a user named "root" and returns all lines that refer to that user.

```
(~) 66% grep root </etc/passwd >search.out 2>search.err
```

Filters, Filenames and Standard Input

Many Unix commands act as filters, taking data from a file or standard input, transforming the data, and writing the results to standard output. Most filters are designed so that if they are called with one or more filenames on the command line, they will use those files as input. Otherwise they will act on standard input. For example, these two commands are equivalent:

```
(~) 66% grep 'gattgc' <big_file.fasta  
(~) 67% grep 'gattgc' big_file.fasta
```

Both commands use the `grep` command to search for the string "gattgc" in the file `big_file.fasta`. The first one searches standard input, which happens to be redirected from the file. The second command is explicitly given the name of the file on the command line.

Sometimes you want a filter to act on a series of files, one of which happens to be standard input. Many filters let you use "-" on the command line as an alias for standard input. Example:

```
(~) 68% grep 'gattgc' big_file.fasta bigger_file.fasta -
```

This example searches for "gattgc" in three places. First it looks in `big_file.fasta`, then in `bigger_file.fasta`, and lastly in standard input (which, since it isn't redirected, will come from the keyboard).

Standard I/O and Pipes

The coolest thing about the Unix shell is its ability to chain commands together into pipelines. Here's an example:

```
(~) 65% grep gattgc big_file.fasta | wc -l  
22
```

There are two commands here. `grep` searches a file or standard input for lines containing a particular string. Lines which contain the string are printed to standard output. `wc -l` is the familiar word count program, which counts words, lines and characters in a file or standard input. The `-l` command-line option instructs `wc` to print out just the line count. The `|` character, which is known as the "pipe" character, connects the two commands together so that the standard output of `grep` becomes the standard input of `wc`.

What does this pipe do? It prints out the number of lines in which the string "gattgc" appears in the file `big_file.fasta`.

More Pipe Idioms

Pipes are very powerful. Here are some common command-line idioms.

Count the Number of Times a Pattern does NOT Appear in a File

The example at the top of this section showed you how to count the number of lines in which a particular string pattern

appears in a file. What if you want to count the number of lines in which a pattern does **not** appear?

Simple. Reverse the test with the **grep -v** switch:

```
(~) 65% grep -v gattgc big_file.fasta | wc -l  
2921
```

Uniquify Lines in a File

If you have a long list of names in a text file, and you are concerned that there might be some duplicates, this will weed out the duplicates:

```
(~) 66% sort long_file.txt | uniq > unique.out
```

This works by sorting all the lines alphabetically and piping the result to the **uniq** program, which removes duplicate lines that occur together. The output is placed in a file named *unique.out*.

Concatenate Several Lists and Remove Duplicates

If you have several lists that might contain repeated entries among them, you can combine them into a single unique list by **cating** them together, then uniquifying them as before:

```
(~) 67% cat file1 file2 file3 file4 | sort | uniq
```

Count Unique Lines in a File

If you just want to know how many unique lines there are in the file, add a **wc** to the end of the pipe:

```
(~) 68% sort long_file.txt | uniq | wc -l
```

Page Through a Really Long Directory Listing

Pipe the output of **ls** to the **more** program, which shows a page at a time. If you have it, the **less** program is even better:

```
(~) 69% ls -l | more
```

Monitor a Rapidly Growing File for a Pattern

Pipe the output of **tail -f** (which monitors a growing file and prints out the new lines) to **grep**. For example, this will monitor the */var/log/syslog* file for the appearance of e-mails addressed to *mzhang*:

```
(~) 70% tail -f /var/log/syslog | grep mzhang
```

Beginning Perl Scripting

Simple scripts, Expressions, Operators, Statements, Variables

Simon Prochnik & Lincoln Stein

Suggested Reading

Learning Perl (6th ed.): Chap. 2, 3, 12,
Unix & Perl to the Rescue (1st ed.): Chap. 4 Chapters 1, 2 & 5 of *Learning Perl*.

Lecture Notes

1. [What is Perl?](#)
2. [Some simple Perl scripts](#)
3. [Mechanics of creating a Perl script](#)
4. [Statements](#)
5. [Literals](#)
6. [Operators](#)
7. [Functions](#)
8. [Variables](#)
9. [Processing the Command Line](#)

Problems

What is Perl?

Perl is a Programming Language

Written by Larry Wall in late 80's to process mail on Unix systems and since extended by a huge cast of characters. The name is said to stand for:

1. Pathologically Eclectic Rubbish Lister
2. Practical Extraction and Report Language

Perl Properties

1. Interpreted Language
2. "Object-Oriented"
3. Cross-platform
4. Forgiving
5. Great for text
6. Extensible, rich set of libraries
7. Popular for web pages
8. Extremely popular for bioinformatics

Other Languages Used in Bioinformatics

C, C++

Compiled languages, hence very fast.
Used for computation (BLAST, FASTA, Phred, Phrap, ClustalW)
Not very forgiving.

Java

Interpreted, fully object-oriented language.
Built into web browsers.
Supposed to be cross-platform, getting better.

Python , Ruby

Interpreted, fully object-oriented language.
Rich set of libraries.
Elegant syntax.
Smaller user community than Java or Perl.

Some Simple Scripts

Here are some simple scripts to illustrate the "look" of a Perl program.

Print a Message to the Terminal

Code:

```
#!/usr/bin/perl
# file: message.pl
use strict;
use warnings;
print "When that Aprill with his shoures soote\n";
print "The droghte of March ath perced to the roote,\n";
print "And bathed every veyne in swich licour\n";
print "Of which vertu engendered is the flour...\n";
```

Output:

```
(~) 50% perl message.pl
When that Aprill with his shoures soote
The droghte of March ath perced to the roote,
And bathed every veyne in swich licour
Of which vertu engendered is the flour...
```

Do Some Math

Code:

```
#!/usr/bin/perl
# file: math.pl
use strict;
use warnings;
print "2 + 2 =", 2+2, "\n";
print "log(1e23)= ", log(1e23), "\n";
print "2 * sin(3.1414)= ", 2 * sin(3.1414), "\n";
```

Output:

```
(~) 51% perl math.pl
2 + 2 =4
log(1e23)= 52.9594571388631
2 * sin(3.1414)= 0.000385307177203065
```

Run a System Command

Code:

```
#!/usr/bin/perl
# file: system.pl
use strict;
use warnings;
system "ls";
```

Output:

```
(~/docs/grad_course/perl) 52% perl system.pl
index.html          math.pl~          problem_set.html~  what_is_perl.h
index.html~         message.pl        simple.html      what_is_perl.h
math.pl             problem_set.html  simple.html~
```

Return the Time of Day

Code:

```
#!/usr/bin/perl
# file: time.pl
use strict;
use warnings;
$time = localtime;
print "The time is now $time\n";
```

Output:

```
(~) 53% perl time.pl
The time is now Thu Sep 16 17:30:02 1999
```

Mechanics of Writing Perl Scripts

Some hints to help you get going.

Creating the Script

A Perl script is just a text file. Use any text (programmer's) editor. *Don't use word processors like Word.*

By convention, Perl script files end with the extension *.pl*.

I suggest Emacs, because it is already installed on almost all Unix machines, but there are many good options: vi, vim, Textwrangler, eclipse

The Emacs text editor has a *Perl mode* that will auto-format your Perl scripts and highlight keywords. Perl mode will be activated automatically if you end the script name with **.pl**.

GUI-based script writing tools (Aquamacs, xemacs, Textwrangler, Eclipse) are easier to use, but you may have to install them yourself.

Let's write a simple perl script. It'll be a simple text file called time.pl and will contain the lines above.

Let's try doing this in emacs

Emacs Essentials

A GUI version is simpler to use e.g. Aquamacs, run it by adding the icon for the application to your Dock then clicking on the icon. You can also run emacs in a Terminal window. Emacs will be installed on almost every Unix system you encounter.

```
(~) 50% emacs
```

The same shortcuts you can use on the command line work in Emacs

e.g.

control-a (^a)
move cursor to beginning of line etc

The most important Emacs-specific commands

control-x control-f (^x ^f)
open a file
control-x control-w (^x ^w)
save as...
control-x control-c (^x ^c)
quit
control-g (^g)
cancel command
shift-control-_ (^_)
Undo typing
control-h ?(^h ?)
Help!!
option-; (M;)
Add comment
option-/ (M/)
Variable/subroutine name auto-completion (cycles through options)

Running the Script

Don't forget to save any changes in your script before running it. The filled red circle at the top left of the emacs GUI window has a dot in it if there are unsaved changes.

Option 1 (quick, not used much)

Run the **perl** program from the command line, giving it the name of the script file to run.

```
(~) 50% perl time.pl  
The time is now Thu Sep 16 18:09:28 1999
```

Option 2 (as shown in examples above)

Put the magic comment

```
#!/usr/bin/perl
```

at the top of your script.

It's really easy to make a mistake with this complicated line and this causes confusing errors (see below). Double check, or copy from a friend who has it working.

And always add

```
use strict;
use warnings;
```

to the top of your script like in the example below

```
#!/usr/bin/perl
# file: time.pl
use strict;
use warnings;
$time = localtime;
print "The time is now $time\n";
```

Now make the script executable with `chmod +x time.pl`:

```
(~) 51% chmod +x time.pl
```

Run the script as if it were a command:

```
(~) 52% ./time.pl
The time is now Thu Sep 16 18:12:13 1999
```

Note that you have to type `./time.pl` rather than `time.pl` because, by default, **bash** does not search the current directory for commands to execute. To avoid this, you can add the current directory (".") to your search PATH environment variable. To do this, create a file in your home directory named `.profile` and enter the following line in it:

```
export PATH=$PATH:.
```

The next time you log in, your path will contain the current directory and you can type `time.pl` directly.

Common Errors

Plan out your script before you start coding. Write the code, then run it to see if it works. Every script goes through a few iterations before you get it right. Here are some common errors:

Syntax Errors

Code:

```
#!/usr/bin/perl
# file: time.pl
use strict;
use warnings;
$time = localtime;
print "The time is now $time\n";
```

Output:

```
(~) 53% time.pl
Can't modify time in scalar assignment at time.pl line 3, near "localtime";
Execution of time.pl aborted due to compilation errors.
```

Runtime Errors

Code:

```
#!/usr/bin/perl
# file: math.pl
use strict;
use warnings;

$six_of_one = 6;
$half_dozen = 12/2;
$result = $six_of_one/($half_dozen - $six_of_one);
print "The result is $result\n";
```

Output:

```
(~) 54% math.pl
Illegal division by zero at math.pl line 6.
```

Forgetting to Make the Script Executable

```
(~) 55% test.pl
test.pl: Permission denied.
```

Getting the Path to Perl Wrong on the #! line

Code:

```
#!/usr/local/bin/pearl
# file: time.pl
use strict;
use warnings;
$time = localtime;
print "The time is now $time\n";
```

```
(~) 55% time.pl
```

```
time.pl: Command not found.
```

This gives a very confusing error message because the command that wasn't found is 'perl' not time.pl

Useful Perl Command-Line Options

You can call Perl with a few command-line options to help catch errors:

-c

Perform a syntax check, but don't run.

-w

Turn on verbose warnings. Same as

```
use warnings;
```

-d

Turn on the Perl debugger.

Usually you will invoke these from the command-line, as in `perl -cw time.pl` (syntax check `time.pl` with verbose warnings). You can also put them in the top line: `#!/usr/bin/perl -w`.

Perl Statements

A Perl script consists of a series of *statements* and *comments*. Each statement is a command that is recognized by the Perl interpreter and executed. Statements are terminated by the semicolon character (;). They are also usually separated by a newline character to enhance readability.

A *comment* begins with the # sign and can appear anywhere. Everything from the # to the end of the line is ignored by the Perl interpreter. Commonly used for human-readable notes. Use comments plentifully, especially at the beginning of a script to describe what it does, at the beginning of each section of your code and for any complex code.

Some Statements

```
$sum = 2 + 2; # this is a statement  
  
$f = <STDIN>; $g = $f++; # these are two statements  
  
$g = $f  
/  
$sum; # this is one statement, spread across 3 lines
```

The Perl interpreter will start at the top of the script and execute all the statements, in order from top to bottom, until it reaches the end of the script. This execution order can be modified by loops and control structures.

Blocks

It is common to group statements into *blocks* using curly braces. You can execute the entire block conditionally, or turn it into a *subroutine* that can be called from many different places.

Example blocks:

```
{ # block starts
```

```

my $EcoRI = 'GAATTC';
my $sequence = <STDIN>;
print "Sequence contains an EcoRI site" if $sequence=~/$EcoRI/;
} # block ends

my $sequence2 = <STDIN>;
if (length($sequence) < 100) { # another block starts
    print "Sequence is too small. Throw it back\n";
    exit 0;
} # and ends

foreach $sequence (@sequences) { # another block
    print "sequence length = ",length($sequence),"\n";
}

```

Literals

A *literal* is a constant value that you embed directly in the program code. You can think of the value as being *literally* in the code. Perl supports both *string literals* and *numeric literals*. A string literal or a numeric literal is a *scalar* i.e. a single value.

Literals cannot be changed. If you want to change the value of some data, it needs to be a *variable*. Much, much more on this coming up, until you're really sick of the whole thing.

String Literals

String literals are enclosed by single quotes ('') or double quotes (""):

```
'The quality of mercy is not strained.'; # a single-quoted string
"The quality of mercy is not strained."; # a double-quoted string
```

The difference between single and double-quoted strings is that variables and certain special escape codes are interpolated into double quoted strings, but not in single-quoted ones. Here are some escape codes:

\n	New line
\t	Tab
\r	Carriage return
\f	Form feed
\a	Ring bell
\040	Octal character (octal 040 is the space character)
\0x2a	Hexadecimal character (hex 2A is the "*" character)
\cA	Control character (This is the ^A character)
\u	Uppercase next character

\l	Lowercase next character
\U	Uppercase everything until \E
\L	Lowercase everything until \E
\Q	Quote non-word characters until \E
\E	End \U, \L or \Q operation

```
"Here goes\n\tnothing!";
# evaluates to:
# Here goes
#     nothing!

'Here goes\n\tnothing!';
# evaluates to:
# Here goes\n\tnothing!

"Here goes \unnothing!";
# evaluates to:
# Here goes Nothing!

"Here \Ugoes nothing\E";
# evaluates to:
# Here GOES NOTHING!

"Alert! \a\al\al";
# evaluates to:
# Alert! (ding! ding! ding!)
```

Putting backslashes in strings is a problem because they get interpreted as escape sequences. To include a literal backslash in a string, double it:

```
"My file is in C:\\Program Files\\Accessories\\wordpad.exe";
# evaluates to: C:\\Program Files\\Accessories\\wordpad.exe
```

Put a backslash in front of a quote character in order to make the quote character part of the string:

```
"She cried \"Oh dear! The parakeet has flown the coop!\"";
# evaluates to: She cried "Oh dear! The parakeet has flown the coop!"
```

Numeric Literals

You can refer to numeric values using integers, floating point numbers, scientific notation, hexadecimal notation, and octal. With some help from the `Math::Complex` module, you can refer to complex numbers as well:

```

123;      # an integer
1.23;     # a floating point number
-1.23;    # a negative floating point number
1_000_000; # you can use _ to improve readability
1.23E45;   # scientific notation
0x7b;      # hexadecimal notation (decimal 123)
0173;      # octal notation (decimal 123)

use Math::Complex; # bring in the Math::Complex module
12+3*i;    # complex number 12 + 3i

```

Backtick Strings

You can also enclose a string in backticks (`). This has the helpful property of executing whatever is inside the string as a Unix system command, and returning its output:

```

`ls -l`;
# evaluates to a string containing the output of running the
# ls -l command

```

Lists

The last type of literal that Perl recognizes is the *list*, which is multiple values strung together using the comma operator (,) and enclosed by parentheses. Lists are closely related to *arrays*, which we talk about later. *Lists* (and *arrays*) are composed from zero, one or more *scalars*, making an empty list, a list containing a single item or a more typical list containing many items, respectively.

```

('one', 'two', 'three', 1, 2, 3, 4.2);
# this is 7-member list contains a mixure of strings, integers
# and floats

```

Operators

Perl has numerous *operators* (over 50 of them!) that perform operations on string and numeric values. Some operators will be familiar from algebra (like "+", to add two numbers together), while others are more esoteric (like the "." string concatenation operator).

Numeric & String Operators

The `.` operator acts on strings. The `!"` operator acts on strings and numbers. The rest act on numbers.

Operator	Description	Example	Result
<code>.</code>	String concatenate	<code>'Teddy' . 'Bear'</code>	<code>TeddyBear</code>
<code>=</code>	Assignment	<code>\$a = 'Teddy'</code>	<code>\$a variable contains 'Teddy'</code>
<code>+</code>	Addition	<code>3+2</code>	<code>5</code>
<code>-</code>	Subtraction	<code>3-2</code>	<code>1</code>
<code>-</code>	Negation	<code>-2</code>	<code>-2</code>
<code>!</code>	Not	<code>!1</code>	<code>0</code>
<code>*</code>	Multiplication	<code>3*2</code>	<code>6</code>
<code>/</code>	Division	<code>3/2</code>	<code>1.5</code>
<code>%</code>	Modulus	<code>3%2</code>	<code>1</code>
<code>**</code>	Exponentiation	<code>3**2</code>	<code>9</code>
<code><FILEHANDLE></code>	File input	<code><STDIN></code>	Read a line of input from standard input
<code>>></code>	Right bit shift	<code>3>>2</code>	<code>0 (binary 11>>2=00)</code>
<code><<</code>	Left bit shift	<code>3<<2</code>	<code>12 (binary 11<<2=1100)</code>
<code> </code>	Bitwise OR	<code>3 2</code>	<code>3 (binary 11 10=11)</code>
<code>&</code>	Bitwise AND	<code>3&2</code>	<code>2 (binary 11&10=10)</code>
<code>^</code>	Bitwise XOR	<code>3^2</code>	<code>1 (binary 11^10=01)</code>

Operator Precedence

When you have an expression that contains several operators, they are evaluated in an order determined by their *precedence*. The precedence of the mathematical operators follows the rules of arithmetic. Others follow a precedence that usually does what you think they should do. If uncertain, use parentheses to force precedence:

```
2+3*4;      # evaluates to 14, multiplication has precedence over addition
(2+3)*4;    # evaluates to 20, parentheses force the precedence
```

Logical Operators

These operators compare strings or numbers, returning TRUE or FALSE:

Numeric Comparison		String Comparison	
<code>3 == 2</code>	equal to	<code>'Teddy' eq 'Bear'</code>	equal to
<code>3 != 2</code>	not equal to	<code>'Teddy' ne 'Bear'</code>	not equal to
<code>3 < 2</code>	less than	<code>'Teddy' lt 'Bear'</code>	less than
<code>3 > 2</code>	greater than	<code>'Teddy' gt 'Bear'</code>	greater than

3 <= 2	less or equal	'Teddy' le 'Bear'	less than or equal
3 >= 2	greater than or equal	'Teddy' ge 'Bear'	greater than or equal
3 <=> 2	compare	'Teddy' cmp 'Bear'	compare
		'Teddy' =~ /Bear/	pattern match

The **<=>** and **cmp** operators return:

- **-1** if the left side is less than the right side
- **0** if the left side equals the right side
- **+1** if the left side is greater than the right side

File Operators

Perl has special *file operators* that can be used to query the file system. These operators generally return TRUE or FALSE.

Example:

```
print "Is a directory!\n" if -d '/usr/home';
print "File exists!\n" if -e '/usr/home/lstein/test.txt';
print "File is plain text!\n" if -T '/usr/home/lstein/test.txt';
```

There are many of these operators. Here are some of the most useful ones:

-e filename	file exists
-r filename	file is readable
-w filename	file is writable
-x filename	file is executable
-z filename	file has zero size
-s filename	file has nonzero size (returns size)
-d filename	file is a directory
-T filename	file is a text file
-B filename	file is a binary file
-M filename	age of file in days since script launched
-A filename	same for access time

Functions

In addition to its operators, Perl has many *functions*. Functions have a human-readable name, such as **print** and take one or more arguments passed as a list. A function may return no value, a single value (AKA "scalar"), or a list (AKA "array"). You can enclose the argument list in parentheses, or leave the parentheses off.

A few examples:

```
# The function is print. Its argument is a string.  
# The effect is to print the string to the terminal.  
print "The rain in Spain falls mainly on the plain.\n";  
  
# Same thing, with parentheses.  
print("The rain in Spain falls mainly on the plain.\n");  
  
# You can pass a list to print. It will print each argument.  
# This prints out "The rain in Spain falls 6 times in the plain."  
print "The rain in Spain falls ",2*4-2," times in the plain.\n";  
  
# Same thing, but with parentheses.  
print ("The rain in Spain falls ",2*4-2," times in the plain.\n");  
  
# The length function calculates the length of a string,  
# yielding 45.  
length "The rain in Spain falls mainly on the plain.\n";  
  
# The split function splits a string based on a delimiter pattern  
# yielding the list ('The','rain in Spain','falls mainly','on the plain.')  
split '/', 'The/rain in Spain/falls mainly/on the plain.';
```

Creating Your Own Functions

You can define your own functions or redefine the built-in ones using the **sub** function. This is described in more detail in the lesson on creating subroutines, which you'll be seeing soon..

Often Used Functions (alphabetic listing)

For specific information on a function, use **perldoc -f function_name** to get a concise summary.

abs	absolute value
chdir	change current directory
chmod	change permissions of file/directory
chomp	remove terminal newline from string variable
chop	remove last character from string variable
chown	change ownership of file/directory
close	close a file handle
closedir	close a directory handle
cos	cosine
defined	test whether variable is defined
delete	delete a key from a hash

die	exit with an error message
each	iterate through keys & values of a hash
eof	test a filehandle for end of file
eval	evaluate a string as a perl expression
exec	quit Perl and execute a system command
exists	test that a hash key exists
exit	exit from the Perl script
glob	expand a directory listing using shell wildcards
gmtime	current time in GMT
grep	filter an array for entries that meet a criterion
index	find location of a substring inside a larger string
int	throw away the fractional part of a floating point number
join	join an array together into a string
keys	return the keys of a hash
kill	send a signal to one or more processes
last	exit enclosing loop
lc	convert string to lowercase
lcfirst	lowercase first character of string
length	find length of string
local	temporarily replace the value of a global variable
localtime	return time in local timezone
log	natural logarithm
m//	pattern match operation
map	perform operation on each member of array or list
mkdir	make a new directory
my	create a local variable
next	jump to the top of enclosing loop
open	open a file for reading or writing
opendir	open a directory for listing
pack	pack a list into a compact binary representation

package	create a new namespace for a module
pop	pop the last item off the end of an array
print	print to terminal or a file
printf	formatted print to a terminal or file
push	push a value onto the end of an array
q/STRING/	generalized single-quote operation
qq/STRING/	generalized double-quote operation
qx/STRING/	generalized backtick operation
qw/STRING/	turn a space-delimited string of words into a list
rand	random number generator
read	read binary data from a file
readdir	read the contents of a directory
readline	read a line from a text file
readlink	determine the target of a symbolic link
redo	restart a loop from the top
ref	return the type of a variable reference
rename	rename or move a file
require	load functions defined in a library file
return	return a value from a user-defined subroutine
reverse	reverse a string or list
rewinddir	rewind a directory handle to the beginning
rindex	find a substring in a larger string, from right to left
rmdir	remove a directory
s///	pattern substitution operation
scalar	force an expression to be treated as a scalar
seek	reposition a filehandle to an arbitrary point in a file
select	make a filehandle the default for output
shift	shift a value off the beginning of an array
sin	sine
sleep	put the script to sleep for a while

sort	sort an array or list by user-specified criteria
splice	insert/delete array items
split	split a string into pieces according to a pattern
sprintf	formatted string creation
sqrt	square root
stat	get information about a file
sub	define a subroutine
substr	extract a substring from a string
symlink	create a symbolic link
system	execute an operating system command, then return to Perl
tell	return the position of a filehandle within a file
tie	associate a variable with a database
time	return number of seconds since January 1, 1970
tr///	replace characters in a string
truncate	truncate a file (make it smaller)
uc	uppercase a string
ucfirst	uppercase first character of a string
umask	change file creation mask
undef	undefine (remove) a variable
unlink	delete a file
unpack	the reverse of pack
untie	the reverse of tie
unshift	move a value onto the beginning of an array
use	import variables and functions from a library module
values	return the values of a hash variable
wantarray	return true in an array context
warn	print a warning to standard error
write	formatted report generation

Ok, now you know all the perl functions, so we're done. Thanks for coming.

Variables

A variable is a symbolic placeholder for a value, a lot like the variables in algebra. These values can be changed. Compare literals whose values cannot be changed. Perl has several built-in variable types:

Scalars: `$variable_name`

A single-valued variable, always preceded by a \$ sign.

Arrays: `@array_name`

A multi-valued variable indexed by integer, preceded by an @ sign.

Hashes: `%hash_name`

A multi-valued variable indexed by string, preceded by a % sign.

Filehandle: `FILEHANDLE_NAME`

A file to read and/or write from. Filehandles have no special prefix, but are usually written in all uppercase.

We discuss arrays, hashes and filehandles later.

Scalar Variables

Scalar variables have names beginning with \$. The name must begin with a letter or underscore, and can contain as many letters, numbers or underscores as you like. These are all valid scalars:

- \$foo
- \$The_Big_Bad_Wolf
- \$R2D2
- \$_____A23
- \$Once_Upon_a_Midnight_Dreary_While_I_Pondered_Weak_and_Weary

You assign values to a scalar variable using the = operator (not to be confused with ==, which is numeric comparison). You read from scalar variables by using them wherever a value would go.

A scalar variable can contain strings, floating point numbers, integers, and more esoteric things. You don't have to predeclare scalars. A scalar that once held a string can be reused to hold a number, and vice-versa:

Code:

```
$p = 'Potato'; # $p now holds the string "potato"
$bushels = 3; # $bushels holds the value 3
$potatoes_per_bushel = 80; # $potatoes_per_bushel contains 80;

$total_potatoes = $bushels * $potatoes_per_bushel; # 240

print "I have $total_potatoes $p\n";
```

Output:

```
I have 240 Potato
```

Scalar Variable String Interpolation

The example above shows one of the interesting features of double-quoted strings. If you place a scalar variable inside a double quoted string, it will be interpolated into the string. With a single-quoted string, no interpolation occurs.

To prevent interpolation, place a backslash in front of the variable:

```
print "I have \$total_potatoes \$p\n";
# prints: I have $total_potatoes $p
```

Operations on Scalar Variables

You can use a scalar in any string or numeric expression like `$hypotenuse = sqrt($x**2 + $y**2)` or `$name = $first_name . ' ' . $last_name`. There are also numerous shortcuts that combine an operation with an assignment:

`$a++`

Increment `$a` by one

`$a--`

Decrement `$a` by one

`$a += $b`

Modify `$a` by adding `$b` to it.

`$a -= $b`

Modify `$a` by subtracting `$b` from it.

`$a *= $b`

Modify `$a` by multiplying `$b` to it.

`$a /= $b`

Modify `$a` by dividing it by `$b`.

`$a .= $b`

Modify the **string** in `$a` by appending `$b` to it.

Example Code:

```
$potatoes_per_bushel = 80; # $potatoes_per_bushel contains 80;
$p = 'one';
$p .= ' ';      # append a space
$p .= 'potato'; # append "potato"

$bushels = 3;
$bushels *= $potatoes_per_bushel; # multiply

print "From $p come $bushels.\n";
```

Output:

From one potato come 240.

String Functions that Come in Handy for Dealing with Sequences

Reverse the Contents of a String

```
$name          = 'My name is Lincoln';
$reversed_name = reverse $name;
print $reversed_name, "\n";
# prints "nlocniL si eman yM"
```

Translating one set of letters into another set

```
$name = 'My name is Lincoln';
# swap a->g and c->t
$name =~ tr/ac/gt/;
print $name, "\n";
# prints "My ngme is Lintoln"
```

Can you see how a combination of these two operators might be useful for computing the reverse complement?

Processing Command Line Arguments

When a Perl script is run, its command-line arguments (if any) are stored in an automatic array called `@ARGV`. You'll learn how to manipulate this array later. For now, just know that you can call the `shift` function repeatedly from the main part of the script to retrieve the command line arguments one by one.

Printing the Command Line Argument

Code:

```
#!/usr/bin/perl
# file: echo.pl
use strict;
use warnings;
$argument = shift;
print "The first argument was $argument.\n";
```

Output:

```
(~) 50% chmod +x echo.pl
(~) 51% echo.pl tuna
The first argument was tuna.
(~) 52% echo.pl tuna fish
The first argument was tuna.
(~) 53% echo.pl 'tuna fish'
The first argument was tuna fish.
(~) 53% echo.pl
The first argument was.
```

Computing the Hypotenuse of a Right Triangle

Code:

```
#!/usr/bin/perl
# file: hypotenuse.pl
use strict;
use warnings;
$x = shift;
$y = shift;
$x>0 and $y>0 or die "Must provide two positive numbers";

print "Hypotenuse=",sqrt($x**2+$y**2), "\n";
```

Output:

```
(~) 82% hypotenuse.pl
Must provide two positive numbers at hypotenuse.pl line 6.
(~) 83% hypotenuse.pl 1
Must provide two positive numbers at hypotenuse.pl line 6.
(~) 84% hypotenuse.pl 3 4
Hypotenuse=5
(~) 85% hypotenuse.pl 20 18
Hypotenuse=26.9072480941474
(~) 86% hypotenuse.pl -20 18
Must provide two positive numbers at hypotenuse.pl line 6.
```

Perl II

Operators, truth, control structures, functions, and
processing the command line

Dave Messina

v3 2012

1

Math

```
1 + 2 = 3      # kindergarten
x      = 1 + 2      # algebra
my $x = 1 + 2;  # Perl
```

What are the differences between the
algebra version and the Perl version?

2

Math

```
my $x = 5;
```

```
my $y = 2;
```

```
my $z = $x + $y;
```

3

Math

```
my $sum = $x + $y;
```

```
my $difference = $x - $y;
```

```
my $product = $x * $y;
```

```
my $quotient = $x / $y;
```

```
my $remainder = $x % $y;
```

4

Math

```
my $x = 5;
```

```
my $y = 2;
```

```
my $sum      = $x + $y;
```

```
my $product = $x - $y;
```

Variable names are arbitrary. Pick good ones!

5

What are these called?

```
my $sum      = $x + $y;
```

```
my $difference = $x - $y;
```

```
my $product = $x * $y;
```

```
my $quotient = $x / $y;
```

```
my $remainder = $x % $y;
```

6

Numeric operators

Operator	Meaning
+	add 2 numbers
-	subtract left number from right number
*	multiply 2 numbers
/	divide left number from right number
%	divide left from right and take remainder
**	take left number to the power of the right number

7

Numeric comparison operators

Operator	Meaning
<	Is left number smaller than right number?
>	Is left number bigger than right number?
<=	Is left number smaller or equal to right?
>=	Is left number bigger or equal to right?
==	Is left number equal to right number?
!=	Is left number not equal to right number?

Why == ?

8

Comparison operators are **yes or no** questions

or, put another way, **true** or **false** questions

True or false:

> Is left number larger than right number?

2 > 1 # true

1 > 3 # false

9

Comparison operators are **true or false** questions

5 > 3

-1 <= 4

5 == 5

7 != 4

10

What is truth?

- 0 the number 0 is false
- "0" the string 0 is false
- "" and '' an empty string is false
- my \$x; an undefined variable is false
- everything else is true

11

Examples of truth

```
my $a;          # FALSE (not yet defined)
$x = 1;         # TRUE
$x = 0;         # FALSE
$x = "\n";      # FALSE
$x = 'true';   # TRUE
$x = 'false';  # TRUE (watch out! "false" is a nonempty string)
$x = ' ';       # TRUE (a single space is non-empty)
$x = "\n\n";    # TRUE (a single newline is non-empty)
$x = 0.0;       # FALSE (converts to string "0")
$x = '0.0';    # TRUE (watch out! The string "0.0" is not the
                 #           same as "0")
```

12

Sidebar: = vs ==

1 equals sign to *make* the left side equal the right side.
2 equals signs to *test* if the left side is equal to the right.

```
my $x;           # x is undefined  
my $x = 1;       # x is now defined  
if ($x == 1)    # is $x equal to 1?  
if ($x = 1)     # (wrong)
```

use warnings will catch this error.

13

Logical operators

Use and and or to combine comparisons.

Operator	Meaning
and	TRUE if left side is TRUE and right side is TRUE
or	TRUE if left side is TRUE or right side is TRUE

14

Logical operator examples

```
if ($i < 100 and $i > 0) {  
    print "$i is the right size\n";  
}  
else {  
    print "out of bounds error!\n";  
}  
  
if ($age < 10 or $age > 65) {  
    print "Your movie ticket is half price!\n";  
}
```

Let's test some more

15

Logical operators

Use not to reverse the truth.

```
$ok = ($i < 100 and $i > 0);  
print "a is too small\n" if not $ok;  
  
# same as this:  
print "a is too small\n" unless $ok;
```

16

defined and undef

defined lets you test whether a variable is defined.

```
if (defined $x) {  
    print "$x is defined\n";  
}
```

undef lets you empty a variable, making it undefined.

```
undef $x;  
print $x if defined $x;
```

17

if not

Testing for defined-ness:

```
if (defined $x) {  
    print "$x is defined\n";  
}
```

What if you wanted to test for undefined-ness?

```
if (not defined $x) {  
    print "x is undefined\n";  
}
```

18

if not

or you could use unless:

```
unless (defined $x) {  
    print "$x is undefined\n";  
}
```

19

Sidebar: operator precedence

Some operators have higher precedence than others.

```
my $result = 3 + 2 * 5;
```

```
# force addition before multiplication  
my $result = (3 + 2) * 5 = 25;
```

The universal precedence rule is this:
multiplication comes before addition,
use parentheses for everything else.

20

String operators

Operator	Meaning
eq	Is the left string same as the right string?
ne	Is the left string not the same as the right string?
lt	Is the left string alphabetically before the right?
gt	Is the left string alphabetically after the right?
.	add the right string to the end of the left string

21

String operator examples

```
my $his_first = 'Barry';
my $his_last  = 'White';
my $her_first = 'Betty';
my $her_last  = 'White';

my $his_full  = $his_first . ' ' . $his_last;
if ($his_last eq $her_last) {
    print "Same\n\n";
}
if ($his_first lt $her_first) {
    print "$his_first before $her_first\n\n";
}
```

22

Comparing numeric and string operators

Numeric	Meaning	String
<code>==</code>	equal to	<code>eq</code>
<code>!=</code>	not equal to	<code>ne</code>
<code>></code>	greater than	<code>gt</code>
<code><</code>	less than	<code>lt</code>
<code>+</code>	addition(concatenation)	<code>.</code>

23

Control structures

Control structures allow you to control if and how a line of code is executed.

You can create alternative branches in which different sets of statements are executed depending on the circumstances.

You can create various types of repetitive loops.

24

Control structures

So far you've seen a basic program,
where every line is executed, in
order, and only once.

```
my $x = 1;  
my $y = 2;  
my $z = $x + $y;  
print "$x + $y = $z\n\n";
```

25

Control structures

Here, the print statement is only
executed some of the time.

```
my $x = 1;  
my $y = 2;  
if ($x == $y) {  
    print "$x and $y are equal\n\n";  
}
```

26

Components of a control structure

1. a keyword

2. a statement in parentheses

3. squiggly brackets

```
if ($x == $y) {  
    print "$x and $y are equal\n\n";  
}
```

The part enclosed by the squiggly brackets is called a **block**.

27

Components of a control structure

When you program, build the structure first and then fill in.

1. a keyword

2. a statement in parentheses

3. squiggly brackets

```
if ($x == $y) {  
    print "$x and $y are equal\n\n";  
}
```

4. now add the print statement

28

if

```
if ($x == $y) {  
    print "$x and $y are equal\n\n";  
}
```

If \$x is the same as \$y, then the print statement will be executed.

or said another way:

If (`$x == $y`) is **true**, then the print statement will be executed.

29

if — a common mistake

```
if ($x = $y) {  
    print "$x and $y are equal\n\n";  
}
```

What happens if we write it this way?

30

else

If the `if` statement is **false**, then the first print statement will be skipped and only the second print statement will be executed.

```
if ($x == $y) {  
    print "$x and $y are equal\n\n";  
}  
else {  
    print "$x and $y aren't equal\n\n";  
}
```

31

elsif

Sometimes you want to test a series of conditions.

```
if ($x == $y) {  
    print "$x and $y are equal\n\n";  
}  
elsif ($x > $y) {  
    print "$x is bigger than $y\n\n";  
}  
elsif ($x < $y) {  
    print "$x is smaller than $y\n\n";  
}
```

32

elsif

What if more than one condition is true?

```
if (1 == 1) {  
    print "$x and $y are equal\n\n";  
}  
elsif (2 > 0) {  
    print "2 is positive\n\n";  
}  
elsif (2 < 10) {  
    print "2 is smaller than 10\n\n";  
}
```

33

given-when

is another way to test a series of conditions
(whose full power you'll learn later).

```
my $x = 3;  
given($x) {  
    when ($x % 2 == 0) {  
        print '$x is even';  
    }  
    when ($x < 10) {  
        print '$x is less than 10';  
    }  
    default {  
        die q(I don't know what to do with $x);  
    }  
}
```

34

unless

It's exactly the opposite of `if (something) *`
These statements are equivalent:

```
if ($x > 0) {  
    print "$x is positive\n\n";  
}  
unless ($x < 0) {  
    print "$x is positive\n\n";  
}
```

If the statement `($x < 0)` is **false**, then the print statement will be executed.

*except you can't `unless..else` or `unless..elsif`

35

while

As long as `($x == $y)` is **true**, the print statement will be executed over and over again.

```
while ($x == $y) {  
    print "$x and $y are equal\n\n";  
}
```

Why might you want to execute a block repeatedly?

36

one line conditionals

An alternative form that sometimes reads better. The conditional comes at the end and parentheses are optional.

```
print "x is less than y\n" if $x < $y;  
print "x is less than y\n" unless $x >= $y;
```

However, you can execute only one statement because there's no longer brackets to enclose multiple lines. Only works for `if` and `unless`.

37

functions

Functions are like operators — they do something with the data you give them. They have a human-readable name, such as `print` and take one or more arguments.

```
print "The rain in Spain falls mainly on the plain.\n\n";
```

38

functions

The function is print. Its argument is a string.
The effect is to print the string to the terminal.

```
print "The rain in Spain falls mainly on the plain.\n\n";
```

39

functions

You can enclose the argument list in parentheses, or leave the parentheses off.

```
# Same thing, with parentheses.  
print("The rain in Spain falls mainly on the plain.\n");
```

40

function examples

You can pass multiple values separated by commas to print, and it will print each argument.

```
# This prints out "The rain in Spain falls 6 times in the plain."  
print "The rain in Spain falls ", 2*4-2, " times in the plain.\n\n";
```

```
# Same thing, but with parentheses.  
print ("The rain in Spain falls ", 2*4-2, " times in the plain.\n");
```

41

functions

A function may return no value, a single value, or multiple values.

```
# print returns nothing.  
print "The rain in Spain falls mainly on the plain.\n\n";  
  
# The length function calculates the length of a string  
# and returns the answer.  
  
my $length = length "The rain in Spain falls mainly on the plain.\n\n";
```

42

processing the command line

Often when you run a program, you want to pass it some information. For example, some numbers, or a filename.

These are called **arguments**.

```
$ add 1 2  
$ parse_blast.pl mydata.blast
```

What are the command-line arguments in these examples?

43

processing the command line

You can give arguments to Perl programs you write, and you can see those arguments inside your script using the shift function.

```
#!/usr/bin/perl  
  
my $arg1 = shift;  
my $arg2 = shift;  
print "my command-line arguments were $arg1 and $arg2\n";
```

44

Perl III

File I/O, more on system calls

Dave Messina

v4 post 2012

1

File I/O

I/O stands for input/output.

It's how computer programs talk
to the rest of the world.

2

Perl has magic

Perl has a magic way that makes it super easy to get data from files and into your program.

It looks like this: <>

3

<>

<> will:

read filenames that are arguments on the command line

open each file in turn

read each line from the file

4

<>

```
#!/usr/bin/perl
# how to read a file with <>
use warnings;
use strict;

while (my $line = <>) {
    chomp $line;
    print "Here's a line: ", $line, "\n";
}
```

5

Sidebar: chomp

chomp removes the newline from the end of a string (if there is a newline).

```
my $string = "hey there!\n";
print "my string is: ", $string, "\n";
chomp $string;
print "after chomp : ", $string, "\n";
```

When you read a file, the first thing you always want to do is chomp.

6

<>

Let's make a file and read from it.
We'll call it myfile.txt

```
% perl read_from_file.pl myfile.txt
```

And now we're giving the name myfile.txt as a command-line argument to our Perl script.

7

<> line count

Let's do something more interesting than printing the line back out. Let's count how many lines there are in the file.

```
my $line_count;
while (my $line = <>) {
    chomp $line;
    $line_count++;
}
print "There are $line_count lines\n";
```

8

Sidebar: increment operators

Yesterday we learned several numeric operators.
Here are a couple more common ones:

`++` the increment operator

```
my $x = 1;  
$x++;      # add 1 to $x
```

```
# exactly the same as  
$x = $x + 1;
```

9

Sidebar: decrement operators

`--` the decrement operator

```
my $x = 1;  
$x--;      # subtract 1 from $x
```

```
# exactly the same as  
$x = $x - 1;
```

10

<> line count

With `++`, we're counting each time we go through the loop.

```
my $line_count;  
while (my $line = <>) {  
    chomp $line;  
    $line_count++;  
}  
print "There are $line_count lines\n";
```

11

<> multiple files

If there is more than one argument, each one is opened and read completely, one after the other.

```
% perl read_from_file.pl myfile.txt another.txt
```

So let's create another file and try it.

12

<> mistakes

Remember how yesterday we had command-line arguments that were numbers?

Does Perl know that the arguments are files?

```
% perl read_from_file.pl 2 9
```

Let's try it and see what happens.

13

the input loop

Let's step back for a moment and think about why <> works. What is while? What is it testing?

```
my $line_count;
while (my $line = <>) {
    chomp $line;
    $line_count++;
}
print "There are $line_count lines\n";
```

14

the input loop

What exactly is going on on this line?

```
while (my $line = <>) {
```

The <> is a function.

It returns a line of input.

We assign that line to a variable, \$line.

While tests that assignment for truth:

"Can we assign a value to \$line?"

15

the input loop

If there is another line in the file, the answer is "yes, we can, it's TRUE."

If we've hit the end of the file, there are no more lines to read, and so the answer is "no", or FALSE.

When the expression in parentheses is false, we exit the loop.

16

the input loop

Once we've exited the loop, the print statement gets executed.

```
my $line_count;  
while (my $line = <>) {  
    chomp $line;  
    $line_count++;  
}  
print "There are $line_count lines\n";
```

17

the input loop

To summarize:

The while loop will read one line of text after another. At the end of input, the <> operator returns undef and the while loop terminates.

Remember that even blank lines in a file are TRUE, because they consist of a single newline character.

18

STDOUT and STDERR

Every Perl script by default has two places it knows where to write to:

STDOUT and STDERR

19

STDOUT and STDERR

STDOUT

Standard output, used to write data out.

Prints to your screen, but can be redirected to a file or other program from the shell using redirection or pipes.

20

STDOUT and STDERR

STDERR

Standard error, used for diagnostic messages.

Also prints to your screen, and also can be redirected to a file or other program from the shell using redirection or pipes.

21

STDOUT and STDERR

You've actually been usually STDOUT all along.
It's the default place where your program's output goes.

When you use print, you're actually writing to STDOUT.

These are equivalent:

```
print "Well, how did I get here?\n";
print STDOUT "Well, how did I get here?\n";
```

22

STDOUT and STDERR

But you can also specify other places to write to.

Like STDERR:

```
print STDOUT "You may ask yourself:\n";
print STDERR "Well, how did I get here?\n";
```

23

STDOUT and STDERR

At first it looks exactly the same as STDOUT, but if we use output redirection on the command line, we can see that the output is actually going to a different place:

```
$ perl test.pl > output.txt
Well, how did I get here?
```

24

open for reading

<> is great, but often you want to read from a specific file. You can do that using open.

```
my $file = shift;  
open(my $filehandle, '<', $file) or die "can't open  
$file: $!\n";
```

25

open

```
my $file = shift @ARGV;  
open(my $filehandle, '<', $file)  
    or die "can't open $file: $!\n";
```

Let's break this down into pieces:

```
my $file = shift @ARGV;
```

reads the filename from the command line.

26

open

```
open(my $filehandle, '<', $file)
```

open is a function, which takes 3 arguments:

The first argument is a filehandle. Filehandles are how you refer to a file within Perl. STDOUT and STDERR are filehandles.

When you open a file yourself, you make your own filehandle and give it a name (here, I chose \$filehandle).

27

open

```
open(my $filehandle, '<', $file)
```

The second argument is a mode. The modes are borrowed from redirection on the command line.

- < for reading from a file
- > for writing to a file

28

open

```
open(my $filehandle, '<', $file)
```

The third argument is the name of a file to open. It can either be a literal name:

```
open(my $filehandle, '<', 'myfile.txt')
```

or a variable containing a filename:

```
open(my $filehandle, '<', $file)
```

Where can you go for more information on open?

29

open or die

```
or die "can't open $file: $!\n";
```

open or die is a Perl idiom. die is a function that exits the program immediately and prints the specified string to STDERR.

Why or? What is being tested for truth?

30

open — \$!

```
or die "can't open $file: $!\n";
```

`$!` is a special Perl variable that contains error messages from the system. If there was a problem with opening your file, there will be an error message in `$!`, and we can include it in our error string.

Let's try it.

31

open for writing

Open also can be used to open files for *writing* by using '`>`' as the second argument to open.

```
my $out = shift @ARGV;
open(my $filehandle, '>', $out)
    or die "can't open $out: $!\n";
```

Now specify that filehandle when you print:

```
print $filehandle "I'm writing to a file!\n";
```

Be careful! If you open an existing file for writing, you will erase everything inside that file!

32

open

You can open more than one file in a script — just give them different filehandles.

```
my $in  = shift @ARGV;
my $out = 'out.txt';
open(my $in_fh, '<', $in) or die "can't open $in: $!\n";
open(my $out_fh, '>', $out) or die "can't open $out: $!\n";
```

33

open

To read from a filehandle line by line, you put the name of the filehandle inside <>, like this:

```
my $in  = shift @ARGV;
open(my $in_fh, '<', $in) or die "can't open $in: $!\n";

while (my $line = <$in_fh>) {
    chomp $line;
    print "This line is from the file $in: $line\n";
}
```

34

a quick word on system

We saw yesterday that there were two ways of executing a command line from within Perl:

```
# with system  
system("sort $file");  
  
# or with backticks  
`sort $file`;
```

35

a quick word on system

With backticks, you can capture the output from the command into a variable:

```
open(my $out_fh, '>', 'sorted.txt')  
    or die "error:$!";  
my $sorted_output = `sort $file`;  
print $out_fh "sorted output:\n", $sorted_output;
```

36

Arrays

Sofia Robb

What is an Array?

- An array is a named list.
- What is a list?
 - ('cat', 'dog', 'narwhal')
- A named list:
 - @animals = ('cat', 'dog', 'narwhal');

Arrays

- Arrays are denoted with '@' symbol

Arrays

- Each element of an array is a scalar variable
 - number
 - letter
 - word
 - sentence
 - `$scalar_variable`

An array of colors.

```
my @colors = ('red', $favorite_color,  
'cornflower blue', 5);
```

Each element of an array can be accessed by its index.

```
my $first  = $colors[0];  
my $second = $colors[1];  
my $third  = $colors[2];  
my $last   = $colors[-1];
```

Each element of the array is a scalar variable therefore we use the '\$' when we refer to an individual element.

```
my @colors = ('red', $favorite_color,  
'cornflower blue', 5);
```

Each element of an array can be accessed by its index.

```
my $first  = $colors[0];  
my $second = $colors[1];  
my $third  = $colors[2];  
my $last   = $colors[-1];
```

A common MISTAKE is to try to access an element in array context (meaning using the '@').

```
my @colors = ('red', $favorite_color,  
'cornflower blue', 5);
```

This is wrong:

```
my $first = @colors[0];
```

This is correct:

```
my $first = $colors[0];
```

Length of an array

```
my @colors = ('red', $favorite_color,  
'cornflower blue', 5);
```

```
my $length = scalar @colors;  
print "$length\n";  
4
```

```
my $length = @colors;  
print "$length\n";  
4
```

Quick print of an array

```
my @colors = ('red', $favorite_color,  
'cornflower blue', 5);  
  
print "@colors";  
red purple cornflower blue 5
```

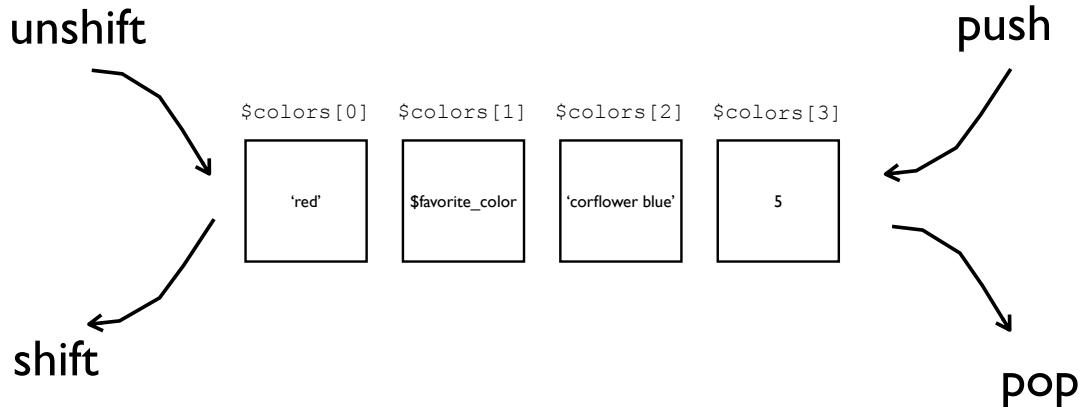
Array to a String

```
my $new_string = join(string , @array);  
  
my @colors = ('red', $favorite_color,  
'cornflower blue', 5);  
  
my $new_string = join ('--' , @colors);  
print "$new_string\n";  
red--purple--cornflower blue--5
```

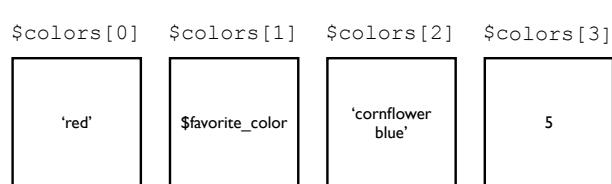
Arrays are Dynamic

```
my @colors = ('red', $favorite_color,  
'cornflower blue', 5);
```

Arrays can grow and shrink



Add elements to the end with `push()`:



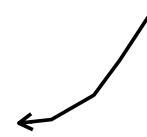
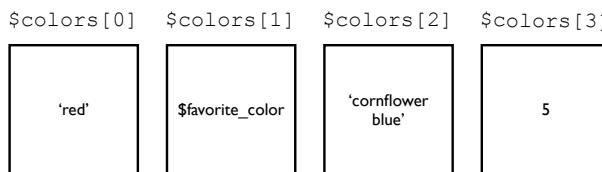
```
push (@array, list of values);
```

```
#add one element to the end  
push (@colors, 'black');
```

```
print join ('--', @colors) , "\n";  
red--purple--cornflower blue--5--black
```

Add elements to the end with push();

push



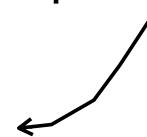
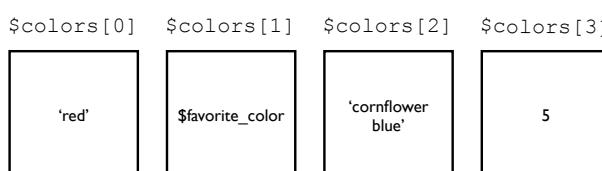
```
push (@array, list of values);
```

```
#add two elements to the end  
push (@colors, 'black' , 'blue');
```

```
print join('---',@colors), "\n";  
red--purple--cornflower blue--5--black--blue
```

Add elements to the end with push();

push



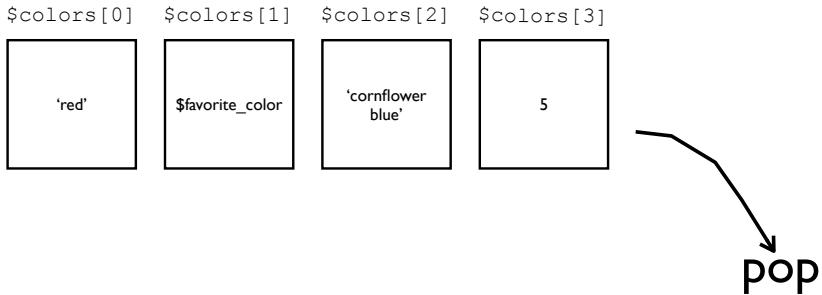
```
push (@array, list of values);
```

```
#add an array of elements  
my @more_colors =  
('yellow', 'pink', 'white', 'orange');
```

```
push (@colors, @more_colors);
```

```
print join('---',@colors) , "\n";  
red--purple--cornflower blue--5--yellow--pink--white--orange
```

Remove an element from the end with pop();



```
my $last_element = pop @colors;  
  
print "$last_element\n";  
5  
print join ('--', @colors) , "\n";  
red--purple--cornflower blue
```

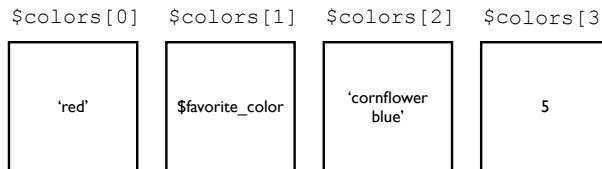
Remove an element from the beginning with shift();



```
my $first_element = shift(@colors);  
  
print "$first_element\n";  
red  
  
print join ('--', @colors) , "\n";  
purple--cornflower blue--5
```

Add elements to the beginning with unshift();

unshift



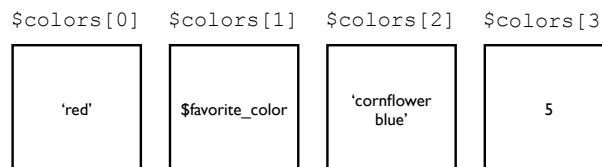
```
unshift (@array, list of values);
```

```
#add one element to the beginning  
unshift (@colors, 'black');
```

```
print join ('--', @colors) , "\n";  
black--red--purple--cornflower blue--5
```

Add elements to the beginning with unshift();

unshift



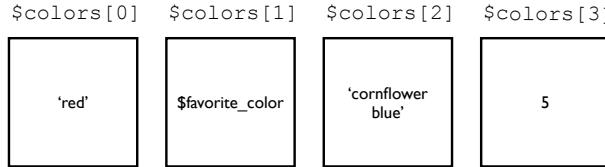
```
unshift (@array, list of values);
```

```
#add one element to the beginning  
unshift (@colors, 'black');
```

```
print join ('--', @colors) , "\n";  
black--red--purple--cornflower blue--5
```

Add elements to the beginning with unshift();

unshift



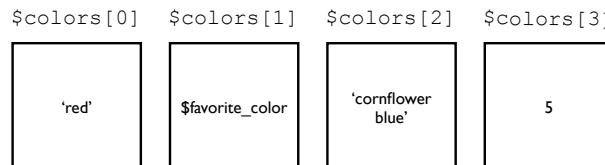
```
unshift (@array, list of values);

#add two elements to the beginning
unshift (@colors, 'black' , 'blue');

print join('---',@colors), "\n";
black--blue--red--purple--cornflower blue
```

Add elements to the beginning with unshift();

unshift



```
unshift (@array, list of values);

#add an array of elements to the beginning
my @more_colors =
('yellow', 'pink', 'white', 'orange');

unshift (@colors, @more_colors);

print join('---',@colors) , "\n";
yellow--pink--white--orange--red--purple--cornflower blue--5
```

Dynamic Arrays

Function	Meaning
push(@array, a list of values)	add value(s) to the end of the list
\$popped_value = pop(@array)	remove a value from the end of the list
\$shifted_value = shift(@array)	remove a value from the front of the list
unshift(@array, a list of values)	add value(s) to the front of the list
splice(...)	everything above and more!

String to an Array

```
my @array = split(pattern , string);

my $string = "I do not like green eggs and ham";
my @words = split(' ', $string);

print join('--', @words), "\n";
I--do--not--like--green--eggs--and--ham
```

Sorting

```
my @words = qw(I do not like green eggs and ham);  
  
my @sorted_words = sort @words;  
  
print join('--' , @sorted_words),"\n";  
I--and--do--eggs--green--ham--like--not  
##ascii sort order. 0-9 then A-Z then a-z
```

Sorting and cmp

```
my @words = qw(I do not like green eggs and ham);  
  
##sort { $a cmp $b} is default sort behavior  
my @sorted_words = sort { $a cmp $b} @words;  
  
print join('--' , @sorted_words),"\n";  
I--and--do--eggs--green--ham--like--not
```

The comparison operator and strings

```
my $x = 'sid';
my $y = 'nancy';
my $result = $x cmp $y;
```

\$result is:

- 1 if the left side is less than the right side
- 0 if the left side equals the right side
- +1 if the left side is greater than the right side

The comparison operator for numbers

```
my $x = 2;
my $y = 3.14;
my $result = $x <=gt; $y;
```

\$result is:

- 1 if the left side is less than the right side
- 0 if the left side equals the right side
- +1 if the left side is greater than the right side

The comparison operator

use cmp to compare two strings

```
my $x = 'sid';
my $y = 'nancy';
my $result = $x cmp $y;
```

use <=> to compare two numbers

```
my $x = 2;
my $y = 3.14;
my $result = $x <=> $y;
```

Numeric Sorting

```
my @numbers = (15,2,10,20,11,1);

## default sorting is ascii
my @sorted_numbers = sort @numbers;
print "@sorted_numbers\n";
1 10 11 15 2 20

@sorted_numbers = sort { $a <=> $b } @numbers;
print "@sorted_numbers\n";
1 2 10 11 15 20
```

Sorting and map

```
my @words = qw(I do not like green eggs and ham);

my @ABC_words = map { uc } @words;
print join('--', @ABC_words), "\n";
I--DO--NOT--LIKE--GREEN--EGGS--AND--HAM

my @sorted_words = sort (@ABC_words);
print join('--', @sorted_words), "\n";
AND--DO--EGGS--GREEN--HAM--I--LIKE--NOT
```

Reverse Sorting

```
my @words = qw(I do not like green eggs and ham);

my @sorted_words = sort { $b cmp $a } @words;
print join('--', @sorted_words), "\n";
not--like--ham--green--eggs--do--and--I
```

Accessing Each Element of an Array

- Loops
 - `foreach`
 - `for`
 - `while`

`foreach` loop

```
## iterate thru @array
foreach my $one_element(@array) {
    ##do something to each $one_element
}
```

foreach

```
my @words = qw(I do not like green eggs and ham);  
  
foreach my $word (@words) {  
    print "$word\n";  
}  
I  
do  
not  
like  
green  
eggs  
and  
ham
```

foreach and sorting

```
my @words = qw(I do not like green eggs and ham);  
  
foreach my $word (sort {uc($a) cmp uc($b)} @words) {  
    print "$word\n";  
}  
and  
do  
eggs  
green  
ham  
I  
like  
not
```

for loop

```
for(initialization; test; increment) {  
    statements;  
}
```

```
for (my $i=0; $i<5 ; $i++) {  
    print "$i\n";  
}  
0  
1  
2  
3  
4
```

for loop

```
for (my $i=0; $i<5 ; $i++) {  
    print "$i\n";  
}
```

	\$i	\$i<5	print "\$i\n";	\$i++
0	0	yes	0	1
1	1	yes	1	2
2	2	yes	2	3
3	3	yes	3	4
4	4	yes	4	5
	5	no		

while loop

```
while(condition) {  
    statements;  
}  
  
my $i = 0;  
while ($i<5) {  
    print "$i\n";  
    $i++;  
}  
0  
1  
2  
3  
4
```

while loop

```
my $i = 0;  
while ($i<5) {  
    print "$i\n";  
    $i++;  
}
```

	\$i	\$i<5	print "\$i\n";	\$i++
0	0	yes	0	1
1	1	yes	1	2
2	2	yes	2	3
3	3	yes	3	4
4	4	yes	4	5
5	5	no		

Loop Control: next

execution of `next()` will cause the loop to jump to the next iteration.

```
my @words = qw(I do not like green eggs and ham);  
  
foreach my $word (sort {uc($a) cmp uc($b)} @words) {  
    next if $word eq 'and';  
    print "$word\n";  
}  
do  
eggs  
green  
ham  
I  
like  
not
```

Loop Control: last

execution of `last()` will cause the loop to exit the loop.

```
my @words = qw(I do not like green eggs and ham);  
  
foreach my $word (sort {uc($a) cmp uc($b)} @words) {  
    print "$word\n";  
    last if $word eq 'and';  
}  
and
```

Example Use of Loops

```
my @seqs = qw(TTT CGG ATG TAA CCC ACC TGA);

my $count = 0;
foreach my $seq (@seqs) {
    if ($seq eq 'TAA' or $seq eq 'TGA' or $seq eq 'TAG') {
        print "*\n";
    } else {
        $count++;
    }
}
print "$count non-stop codons\n";
```

@ARGV holds command line arguments

```
./sample_usr_input.pl 5 five
```

```
print "@ARGV\n";

print "\$ARGV[0]: $ARGV[0]\n";
print "\$ARGV[1]: $ARGV[1]\n";

my $arg1 = shift;
my $arg2 = shift;

print "arg1: $arg1\n";
print "arg2: $arg2\n";
```

```
5 five
$ARGV[0]: 5
$ARGV[1]: five
arg1: 5
arg2: five
```

Hashes

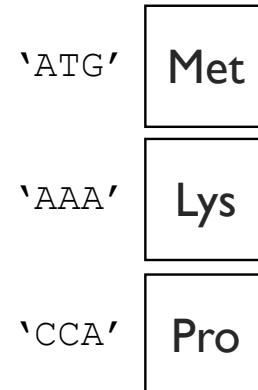
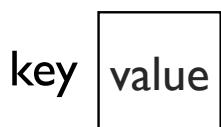
Sofia Robb

Hashes

- Perl hashes are denoted with a '%' symbol like this
%data
- Each key and each value contains a scalar value for example this could be
 - a number
 - a letter
 - a word
 - a sentence
 - a scalar variable like \$scalar_variable
 - a gene ID
 - a sequence

What is a hash?

- A hash is an associative array made up of key/value pairs.
- Like a dictionary
- And unlike an array a hash is unordered.

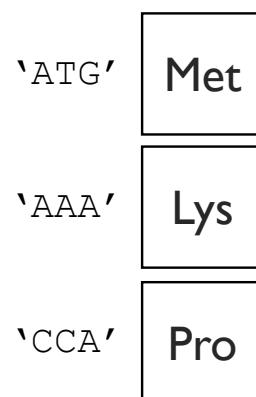


A key is like a descriptive array index.

An array

\$colors[0]	\$colors[1]	\$colors[2]	\$colors[3]
'red'	\$favorite_color	'cornflower blue'	5

A hash



The array index [0] is similar to the key 'ATG'.

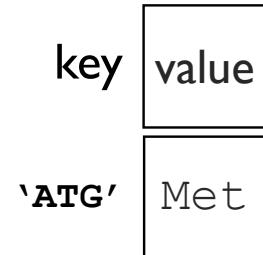
The key 'ATG' is used to access the value 'Met', just as [0] is used to access 'red'

But the key/value pairs are not stored in order

Creating a hash

The hash %genetic_code is built with key/value pairs

```
my %genetic_code = (  
    "ATG" => "Met",  
    "AAA" => "Lys",  
    "CCA" => "Pro",  
) ;
```



Accessing a hash value using a key

```
my %genetic_code = (  
    "ATG" => "Met",  
    "AAA" => "Lys",  
    "CCA" => "Pro",  
) ;  
  
my $aa = $genetic_code{"ATG"};  
print "ATG translates to $aa\n";  
ATG translates to Met
```

Each value of the hash is a scalar therefore we use the '\$' when we refer to an individual value.

Hash keys are surrounded by squiggly brackets {}

keys() returns an unordered list of the keys of a hash

```
@array_of_keys = keys (%hash);  
  
my %genetic_code = (  
    "ATG" => "Met",  
    "AAA" => "Lys",  
    "CCA" => "Pro",  
);  
  
my @codons = keys (%genetic_code);  
print join(" -- ", @codons), "\n";  
CCA -- AAA -- ATG
```

Iterating through a hash by looping through an list of hash keys.

```
my %genetic_code = (  
    "ATG" => "Met",  
    "AAA" => "Lys",  
    "CCA" => "Pro",  
);  
  
foreach my $codon (keys %genetic_code) {  
    my $aa = $genetic_code{$codon};  
    print "$codon translates to $aa\n";  
}  
CCA translates to Pro  
AAA translates to Lys  
ATG translates to Met
```

Remember: the key is used to access
the value
\$value = \$hash{\$key}

Sorting and iterating through the keys of a hash

```
my %genetic_code = (  
    "ATG" => "Met",  
    "AAA" => "Lys",  
    "CCA" => "Pro",  
) ;
```

Remember: hash keys are
unordered so we use `sort` to be
sure that the order is always the
same.

```
foreach my $codon (sort keys %genetic_code) {  
    my $aa = $genetic_code{$codon};  
    print "$codon translates to $aa\n";  
}  
AAA translates to Lys  
ATG translates to Met  
CCA translates to Pro
```

Iterating through a hash and sorting by the values

```
my %genetic_code = (  
    "ATG" => "Met",  
    "AAA" => "Lys",  
    "CCA" => "Pro",  
) ;
```

Remember: the key is used to access
the value
`$value = $hash{$key}`

```
foreach my $codon (sort {$genetic_code{$a} cmp $genetic_code{$b}}  
keys %genetic_code) {  
    my $aa = $genetic_code{$codon};  
    print "$codon translates to $aa\n";  
}  
AAA translates to Lys  
ATG translates to Met  
CCA translates to Pro
```

we can create a custom
sort function using `{$a cmp
$b}`

values() returns an unordered list of values

```
@array_of_values = values(%hash);
```

```
my %genetic_code = (
    "ATG" => "Met",
    "AAA" => "Lys",
    "CCA" => "Pro",
);
```

You can use `sort values` to be
sure that the order of the values is
always the same.

```
my @amino_acids = values(%genetic_code);
print join(" -- ", @amino_acids), "\n";
Pro -- Lys -- Met
```

Adding additional key/value pairs

```
my %genetic_code = (
    "ATG" => "Met",
    "AAA" => "Lys",
    "CCA" => "Pro",
);
```

```
$genetic_code{"TGT"} = "Cys";

foreach my $codon (keys %genetic_code) {
    print "$codon -- $genetic_code{$codon}\n";
}
CCA -- Pro
AAA -- Lys
ATG -- Met
TGT -- Cys
```

Deleting key/value pairs

```
my %genetic_code = (
    "ATG" => "Met",
    "AAA" => "Lys",
    "CCA" => "Pro",
);

delete $genetic_code{ "AAA"} ;

foreach my $codon (keys %genetic_code) {
    print "$codon -- $genetic_code{$codon}\n";
}
CCA -- Pro
ATG -- Met
```

Use exists() to test if a key exists.

```
my %genetic_code = (
    "ATG" => "Met",
    "AAA" => "Lys",
    "CCA" => "Pro",
);
```

key exists?	return value
yes	1
no	'' empty string is false

```
my $codon = "ATG";
if (exists $genetic_code{$codon}) {
    print "$codon -- $genetic_code{$codon}\n";
} else{
    print "key: $codon does not exist\n";
}
ATG -- Met
##when $codon= "TTT", code prints "key: TTT does not exist"
```

Auto increment hash values

Auto increment scalars:

```
my $num = 1;  
print $num , "\n";    #prints 1  
$num++;             #same as $num=$num +1;  
print $num , "\n";    #prints 2
```

Auto increment hash values:

```
my %hash;  
$hash{books} = 0;  
print $hash{books}, "\n";    #prints 0  
$hash{books}++; #same as $hash{books} = $hash{books} + 1  
print $hash{books}, "\n"; #prints 1
```

nothing + 1 equals 1

```
my %hash;  
$hash{books} = 0;  
print $hash{books}, "\n";  
  
$hash{books}++;  
print $hash{books} , "\n"; # prints 1
```

When we first start, the key 'books' doesn't exist.
We try to add 1 to nothing, so the total is 1.

Using hashes for keeping count

```
my $seq = "ATGGGCGTATGCAATT";
my @nucs = split "", $seq;
print "@nucs\n";
#A T G G G C G T A T G C A A T T

my %nt_count;
foreach my $nt (@nucs) {
    $nt_count{$nt}++;
}

foreach my $nt (keys %nt_count) {
    my $count = $nt_count{$nt};
    print "$nt\t$count\n";
}

A      4
T      5
C      2
G      5
```

Creating a hash from variable input like data from a file

```
my $file = shift;
open (my $in_file, '<', $file)
    or die "can't open file $file $!\n";
my %hash;
while (my $line = <$in_file>){
    chomp $line;
    my ($key, $value) = split /\t/, $line;
    $hash{$key} = $value;
}
foreach my $key (sort keys %hash){
    my $value = $hash{$key};
    print "key:$key value:$value\n";
}
```

Regular Expressions

Sofia Robb

What is a regular expression?

A regular expression is a string template against which you can match a piece of text.

They are something like shell wildcard expressions, but **much** more powerful.

Examples of Regular Expressions

This bit of code loops through @ARGV files or STDIN. Finds all lines containing an EcoRI site, and bumps up a counter:

```
my $sites = 0;
while (my $line = <>) {
    chomp $line;
    if ($line =~ /GAATTC/) {
        print "Found an EcoRI site!\n";
        $sites++;
    }
}
print "$sites EcoRI sites total.\n"
```

Examples of Regular Expressions

This does the same thing, but counts one type of methylation site (Pu-C-X-G) instead:

```
my $sites = 0;
while (my $line = <>) {
    chomp $line;
    if ($line =~ /[GA]C.?\w/) {
        print "Found a methylation site!\n";
        $sites++;
    }
}
print "$sites methylation sites total.\n"
```

Specifying the String to Search

To specify which string variable to search, use the `=~` operator:

```
my $h = "Who's afraid of Virginia Woolf?";  
print "I'm afraid!\n" if $h =~ /Woo?lf/;
```

Regular Expression Atoms

A regular expression is normally delimited by two slashes ("/"). Everything between the slashes is a pattern to match. A pattern is composed of one or more atoms:

1. Ordinary characters:

a-z, A-Z, 0-9 and some punctuation.
These match themselves.

2. The `.` character:

matches everything except the newline.

3. A bracket list of characters

[AaGgCcTtNn], [A-F0-9], or [^A-Z]
(the last means anything BUT A-Z).

4. Predefined character sets:

\d The digits [0-9]
\w A word character [A-Za-z_0-9]
\s White space [\t\n\r]
\D A non-digit
\W A non-word
\S Non-whitespace

5. Anchors:

^ Matches the beginning of the string
\$ Matches the end of the string
\b Matches a word boundary (between a \w and a \W)

Regular Expression Atoms

Examples

- `/g..t/` matches "gaat", "goat", and "gotta get a goat" (twice)
- `/g[gatc][gatc]t/` matches "gaat", "gttt", "gatt", and "gotta get an agatt" (once)
- `/\d\d\d-\d\d\d\d/` matches 376-8380, and 5128-8181, but not 055-98-2818.
- `/^\d\d\d-\d\d\d\d/` matches 376-8380 and 376-83801, but not 5128-8181.
- `/^\d\d\d-\d\d\d\d$/` only matches telephone numbers.
- `/\bcat/` matches "cat", "catsup" and "more catsup please" but not "scat".
- `/\bcat\b/` only text containing the word "cat".

Quantifiers

By default, an atom matches once. This can be modified by following the atom with a quantifier:

?	atom matches zero or exactly once
*	atom matches zero or more times
+	atom matches one or more times
{3}	atom matches exactly three times
{2,4}	atom matches between two and four times, inclusive
{4,}	atom matches at least four times

Examples:

- `/goa?t/` matches "goat" and "got". Also any text that contains these words.
- `/g.+t/` matches "goat", "goot", and "grant", among others.
- `/g.*t/` matches "gt", "goat", "goot", and "grant", among others.
- `/^\d{3}-\d{4}$/,` matches US telephone numbers (no extra text allowed.)

Alternatives and Grouping

A set of alternative patterns can be specified with the | symbol:

```
/wolf|sheep/;  
# matches "wolf" or "sheep"  
  
/big bad (wolf|sheep)/;  
# matches "big bad wolf"  
#       or "big bad sheep"
```

Parenthesis and Quantifiers

You can combine parenthesis and quantifiers to quantify entire subpatterns:

```
/Who's afraid of the big (bad )?wolf\?/;  
  
# matches "Who's afraid of the big bad wolf?"  
# and      "Who's afraid of the big wolf?"
```

This also shows how to literally match the special characters -- put a backslash (\) in front of them.

What about finding strings that don't contain the pattern?

use `!~` instead of `=~`

This is equivalent to "not match" operator `!~`, which reverses the sense of the match:

```
$h = "Who's afraid of Virginia Woolf?";  
print "I'm not afraid!\n" if $h !~ /Woo?lf/;
```

Matching with a Variable Pattern

You can use a scalar variable for all or part of a regular expression.

```
$pattern = '/usr/local';  
if ($file =~ /^$pattern/){  
    print "matches" ;  
}
```

See the [o flag](#) for important information about using variables inside patterns.

Subpatterns

You can extract and manipulate subpatterns in regular expressions.

To designate a subpattern, surround its part of the pattern with parenthesis (same as with the grouping operator). This example has just one subpattern, (.+) :

```
/Who's afraid of the big bad w(.+)\?/
```

Using Subpatterns inside the Match

Once a subpattern matches, you can refer to it later within the same regular expression.

The first subpattern becomes \1, the second \2, the third \3, and so on.

Using Subpatterns Inside the Match

```
while (my $line = <>) {  
    chomp $line;  
    if ($line =~ /Who's afraid of the big bad w(.)\1f/) {  
        print "I'm scared!\n"  
    }  
}
```

This loop will print "I'm scared!" for the following matching lines:

- Who's afraid of the big bad woof
 - Who's afraid of the big bad weef
 - Who's afraid of the big bad waaf
- but not
- Who's afraid of the big bad wolf
 - Who's afraid of the big bad wife

Using Subpatterns Inside the Match

```
/\b(\w+)s love \1 food\b/
```

will match "dogs love dog food", but not "dogs love monkey food".

Using Subpatterns Outside the Match

Outside the regular expression match statement, the matched subpatterns (if any) can be found the variables **\$1**, **\$2**, **\$3**, and so forth.

Example. Extract 50 base pairs upstream and 25 base pairs downstream of the TATTAT consensus transcription start site:

```
while (my $line = <>) {  
    chomp $line;  
    next unless $line =~ /(\.{50})TATTAT(\.{25})/;  
    my $upstream = $1;  
    my $downstream = $2;  
}
```

Extracting and Saving Subpatterns Using Arrays

If you assign a regular expression match to an **array**, it will return a list of all the subpatterns that matched. Alternative implementation of previous example:

```
while (my $line = <>) {  
    chomp $line;  
    my ($upstream,$downstream) = $line =~ /(\.{50})TATTAT(\.{25})/;  
}
```

If the regular expression doesn't match at all, then it returns an empty list. Since an empty list is **FALSE**, you can use it in a logical test:

```
while (my $line = <>) {  
    chomp $line;  
    next unless my ($upstream,$downstream) = $line =~ /(\.{50})TATTAT(\.{25})/;  
    print "upstream = $upstream\n";  
    print "downstream = $downstream\n";  
}
```

Grouping without Making Subpatterns

Because parentheses are used both for grouping (`a|ab|c`) and for matching subpatterns, you may match subpatterns that don't want to. To avoid this, group with `(?:pattern)`:

```
/big bad (?:wolf|sheep) /;
```

```
# matches "big bad wolf" or "big bad sheep",
# but doesn't extract a subpattern.
```

Subpatterns and Greediness

By default, regular expressions are "greedy". They try to match as much as they can. For example:

```
$h = 'The fox ate my box of doughnuts';
$h =~ /(f.+x)/;
$subpattern = $1;
```

Because of the greediness of the match, **\$subpattern** will contain "fox ate my box" rather than just "fox".

To match the minimum number of times, put a `?` after the quantifier, like this:

```
$h = 'The fox ate my box of doughnuts';
$h =~ /(f.+?x)/;
$subpattern = $1;
```

Now **\$subpattern** will contain "fox". This is called *lazy* matching.
Lazy matching works with any quantifier, such as `+?, *?, ??` and `{2,50}?`.

String Substitution

The s/// Function

String substitution allows you to replace a pattern or character range with another one using the **s///** and **tr///** functions.

s/// has two parts: the regular expression and the string to replace it with: *s/**expression/replacement/*.

```
$h = "Who's afraid of the big bad wolf?";  
$i = "He had a wife.";  
  
$h =~ s/w.+f/goat/;  
# yields "Who's afraid of the big bad goat?"  
  
$i =~ s/w.+f/goat/;  
# yields "He had a goate."
```

Extract pattern matches and use them in the replacement part of the substitution:

```
$h = "Who's afraid of the big bad wolf?";  
  
$h =~ s/(\w+) (\w+) wolf/$2 $1 wolf/;  
# yields "Who's afraid of the bad big wolf?"
```

Using a Variable in the Substitution Part

```
$h = "Who's afraid of the big bad wolf?";  
  
$animal = 'hyena';  
$h =~ s/(\w+) (\w+) wolf/$2 $1 $animal/;  
# yields "Who's afraid of the bad big hyena?"
```

Translating Character Ranges

The tr/// Function

The **tr///** function allows you to translate one set of characters into another. Specify the source set in the first part of the function, and the destination set in the second part:

```
$h = "Who's afraid of the big bad wolf?";  
  
$h =~ tr/ao/AO/;  
# yields "WhO's AfrAid Of the big bAd wOlF?";
```

This example counts N's in a series of DNA sequences:

tr/// returns the number of characters transformed, which is sometimes handy for counting the number of a particular character without actually changing the string.

Code:

```
while (my $line = <>) {  
    chomp $line;    # assume one sequence per line  
    my $count = $line =~ tr/Nn/Nn/;  
    print "Sequence $line contains $count Ns\n";  
}
```

Input:

```
AGCTGGGAAAGT  
AGCNNNAAAGT  
TAGCNGTTAAAT  
GAATCAGCTGGG  
...
```

Output:

```
(~) 50% count_Ns.pl  
sequence_list.txt  
Sequence 1 contains 0 Ns  
Sequence 2 contains 3 Ns  
Sequence 3 contains 1 Ns  
Sequence 4 contains 0 Ns  
...
```

Common Regular Expression Modifiers

Regular expression matches and substitutions have a whole set of options which you can use by appending one or more modifiers to the end of the operation.

i

Case insensitive match.

g

Global match.

Case insensitive Matches

```
my $string = 'Big Bad WOLF!';
if ($string =~ /wolf/i){
    print "There's a wolf in the closet!";
}
```

Global Matches

Adding the `g` modifier to the pattern causes the match to be global. Called in a scalar context (such as an `if` or `while` statement), it will match as many times as it can.

This will match all codons in a DNA sequence, printing them out on separate lines:

Code:

```
my $sequence = 'GTTGCCTGAAATGGCGGAACCTTGAA';
while ( $sequence =~ /(\.{3})/g ) {
    print $1, "\n";
}
```

Output:

GTT
GCC
TGA
AAT
GGC
GGA
ACC
TTG

The `pos()` function retrieves the position where the next attempt begins

```
$position_of_next_attempt = pos($sequence)
```

If you perform a global match in a **list** context (e.g. assign its result to an array), then you get a list of all the subpatterns that matched from left to right.

This code fragment gets arrays of codons in three reading frames:

```
@frame1 = $sequence =~ /(.{3})/g;
@frame2 = substr($sequence,1) =~ /(.{3})/g;
@frame3 = substr($sequence,2) =~ /(.{3})/g;
```

Additional regular expression modifiers

o

Only compile variable patterns once.

m

Treat string as multiple lines. ^ and \$ will match at start and end of internal lines, as well as at beginning and end of whole string. Use \A and \Z to match beginning and end of whole string when this is turned on.

s

Treat string as a single line. "." will match any character at all, including newline.

x

Allow extra whitespace and comments in pattern.

Subroutines

```
#!/usr/bin/perl

use strict;
use warnings;

my $seq1 = "ac ggTtAa";
my $seq2 = "tTcC aaA tgg";

# clean up $seq1
# 1) make it all lower case
$seq1 = lc $seq1;
# 2) remove white space
$seq1 =~ s/\s//g;

# clean up $seq2
# 1) make it all lower case
$seq2 = lc $seq2;
# 2) remove white space
$seq2 =~ s/\s//g;

# print cleaned up sequences
print "seq1: $seq1\n";
print "seq2: $seq2\n";
```

Problems With This Code

- The same cleanup statements are run for \$seq1 and \$seq2.
- Duplication of code (BAD!).
- Subroutines to the rescue.

Subroutines

- Blocks of code that you can call in different places.
- Code resides in one place.
 - Only need to write the code once.
 - Easier to maintain.
- Take arguments and return results.
- Make code easier to read.
- Like a mini-program within your program.

Creating a Subroutine

I.Turn the code of interest into a block.

```
{  
    # clean up $seq  
    # 1) make it all lower case  
    $seq = lc $seq;  
    # 2) remove white space  
    $seq =~ s/\s//g;  
}
```

Creating a subroutine

2. Label the block with: `sub subroutine_name`

```
sub cleanup_sequence {  
    # clean up $seq  
    # 1) make it all lower case  
    $seq = lc $seq;  
  
    # 2) remove white space  
    $seq =~ s/\s//g;  
}
```

Creating a Subroutine

3. Add statements to read the subroutine argument(s) and return the subroutine result(s).

```
sub cleanup_sequence {  
  
    # get the sequence argument to the  
    # subroutine - note that just like shift gets  
    # an argument for your program, shift gets an  
    # argument to your subroutine  
    my $seq = shift;  
  
    # clean up $seq  
  
    # 1) make it all lower case  
    $seq = lc $seq;  
    # 2) remove white space  
    $seq =~ s/\s//g;  
  
    # return cleaned up sequence  
    return $seq;  
  
}
```

Passing Arguments to a Subroutine

- Arguments are passed in `@_` a special array created by Perl.
 - Analogous to `@ARGV` for program arguments.
 - Can use `shift` to take one argument at a time.

```
# take the first argument  
my $arg1 = shift;  
# take the second argument  
my $arg2 = shift;
```

Passing Arguments to a Subroutine

- Can copy the contents of `@_` into a list of named variables.

```
my ($arg1, $arg2) = @_;
```

Returning Subroutine Results

Use return operator to return results.

- Usually return at the end of the subroutine but can use it to exit the subroutine earlier.
- Return a single value.

```
return $single_value; #scalar
```

- Return a list.

```
return ($variable, "string", 3); #list  
return @array_of_values; #array
```

Returning Subroutine Results

- Return an empty list or undef depending on context.

```
return; #empty list or undef
```

Calling a Subroutine

Calling our subroutine is just like calling an existing built-in Perl function.

```
my $result = my_sub($arg1, $arg2, $arg3, ...);
```

Location of Subroutines

Usually at the bottom of the script.

- Allows to visually separate main program form the subroutines.

```
#!/usr/bin/perl
use strict;
use warnings;

my $seq1 = "ac ggTtAa";
my $seq2 = "tTcC aaA tgg";

# call cleanup_sequence for each sequence
$seq1 = cleanup_sequence($seq1);
$seq2 = cleanup_sequence($seq2);
# print cleaned up sequences
print "seq1: $seq1\n";
print "seq2: $seq2\n";

sub cleanup_sequence {
    # get the sequence argument
    my $seq = shift;
    # cleanup $seq
    # 1) make it all lower case
    $seq = lc $seq;
    # 2) remove white space
    $seq =~ s/\s//g;
    # return cleaned up sequence
    return $seq;
}
```

Scope

```

#!/usr/bin/perl
use strict;
use warnings;
my $x = 100;
my $y = 20;

if ($x > $y) {
    my $z = 10;
    $x = 30;
    print "x (inside if block): $x\n";
    print "y (inside if block): $y\n";
    print "z (inside if block): $z\n";
}

print "x (outside if block): $x\n";
print "y (outside if block): $y\n";
print "z (outside if block): $z\n";

```

Blocks

- That's because `$z` was declared inside the if block, so it's only accessible inside that block.
- Any time we see `{ }`, we're creating a block.
- Blocks are like boxes that have one way mirrors – you can see outside the box from inside, but not inside the box from the outside.
- To fix that error, we need to declare `$z` outside the if block.

Blocks

- Variables declared inside of a block only exist inside the block – once the block is finished, they will be destroyed.

```
#!/usr/bin/perl

use strict;
use warnings;

my $x = 100;
my $y = 20;
my $z = 5;

if ($x > $y) {
    my $z = 10;
    $x = 30;
    print "x (inside if block): $x\n";
    print "y (inside if block): $y\n";
    print "z (inside if block): $z\n";
}

print "x (outside if block): $x\n";
print "y (outside if block): $y\n";
print "z (outside if block): $z\n";
```

Output:
\$x (inside of block):30
\$y (inside of block): 20
\$z (inside of block):10
\$x (outside if block): 30
\$y (outside if block): 20
\$z (outside if block): 5

Scope

Does the program give the expected behavior?

- By declaring “`my $z =10;`” inside the if block, we’re creating a new variable called `$z` only accessible within the block.
- This new variable will not modify the outside variable!
- Note that we can create a new `$z` variable inside the block with no problems – if we do it outside, we’ll get a warning.

Scope

- If we remove “`my`” from that line, the modification to `$z` will show outside the block.

```
#!/usr/bin/perl

use strict;
use warnings;

my $x = 100;
my $y = 20;
my $z = 5;

if ($x > $y) {
    $z = 10;
    $x = 30;
    print "x (inside if block): $x\n";
    print "y (inside if block): $y\n";
    print "z (inside if block): $z\n";
}

print "x (outside if block): $x\n";
print "y (outside if block): $y\n";
print "z (outside if block): $z\n";
```

Output:

```
$x (inside if block): 30
$y (inside if block): 20
$z (inside if block): 10
$x (outside if block): 30
$y (outside if block): 20
$z (outside if block): 10
```

Using Modules

v2 2012

1

Why use modules?

Sometimes you may want to use the same functions over and over again in different programs

Bad way: Copy and paste a subroutine

Good way: Make a module

There are also many many modules that other people have written that you can use!

To use modules they must be properly installed and called with the “use” command

2

File::Basename

basename

Input = long UNIX path name
i.e. '/home/dave/dna.fa'

Output = file name
i.e. 'dna.fa'

dirname

Input = long UNIX path name
i.e. '/home/dave/dna.fa'

Output = directory
'/home/dave/'

Let's try it.

3

File::Basename

```
#!/usr/bin/perl
# file: basename.pl

use strict;
use File::Basename;

my $path = '/home/dave/dna.fa';
my $base = basename($path);
my $dir  = dirname($path);

print "The base is $base and the directory is $dir.\n";
```

Output: The base is dna.fa and the directory is /home/dave.

Common error: Undefined subroutine &main::basename called at basename.pl line 8.

4

Env

This standard module imports a set of scalar variables that describe your environment

```
$HOME  
$PATH  
$USER
```

5

Env

```
#!/usr/bin/perl  
# file env.pl  
  
use strict;  
use Env;  
  
print "My home is $HOME\n";  
print "My path is $PATH\n";  
print "My username is $USER\n";
```

Output:

```
My home is /home/dave  
My path is /home/dave/pfb2012  
My username is dave
```

6

Which modules are installed?

dave\$ perldoc perlmodlib

Which modules are installed with basic perl installation?

<http://perldoc.perl.org/perlmodlib.html>

dave\$ perldoc perllocal

Which modules are installed on your machine?

7

Installing modules manually

```
% tar zxvf bioperl-1.6.1.tar.gz
bioperl-1.6.1/
bioperl-1.6.1/Bio/
...
% perl Makefile.PL
Generated sub tests. go make show_tests to see available subtests
...
Writing Makefile for Bio

% make
cp Bio/Tools/Genscan.pm blib/lib/Bio/Tools/Genscan.pm
...
Manifying blib/man3/Bio::Location::CoordinatePolicyI.3
Manifying blib/man3/Bio::SeqFeature::Similarity.3

% make test
PERL_DL_NONLAZY=1 /net/bin/perl -Iblib/arch -Iblib/lib
-I/net/lib/perl5/5.6.1/i686-linux -I/net/lib/perl5/5.6.1 -e 'use
Test::Harness qw(&runtests $verbose); $verbose=0; runtests @ARGV;' t/*.t
t/AACchange.....ok
...
All tests successful, 95 subtests skipped.
Files=60, Tests=1011, 35 wallclock secs (25.47 cusr + 1.60 csys = 27.07 CPU)

% make install
Installing /net/lib/perl5/site_perl/5.6.1/bioback.pod
Installing /net/lib/perl5/site_perl/5.6.1/biostart.pod
...
```

8

Installing Modules Using the CPAN Shell

Perl has a CPAN module installer built into it. You run it like this:

```
% sudo cpan

cpan shell -- CPAN exploration and modules installation (v1.59_54)
ReadLine support enabled

cpan>
From this shell, there are commands for searching for modules, downloading them, and installing them.

[The first time you run the CPAN shell, it will ask you a lot of configuration questions. Generally, you can just hit return to accept the defaults. The only trick comes when it asks you to select CPAN mirrors to download from. Choose any ones that are in your general area on the Internet and it will work fine.]
```

To search for a module:

```
cpan> i /Wrap/
Going to read /bush_home/bush1/lstein/.cpan/sources/authors/01mailrc.txt.gz
CPAN: Compress::Zlib loaded ok
Going to read /bush_home/bush1/lstein/.cpan/sources/modules/02packages.details.gz
  Database was generated on Tue, 16 Oct 2001 22:32:59 GMT
...
Module      Text::Wrap      (M/MU/MUIR/modules/Text-Tabs+Wrap-2001.0929.tar.gz)
...
41 items found

cpan> install Text::Wrap
Running install for module Text::Wrap

quit
```

9

Where are modules installed?

Module files end with the extension .pm. If the module name is a simple one, like **Env**, then Perl will look for a file named **Env.pm**. If the module name is separated by :: sections, Perl will treat the :: characters like directories. So it will look for the module **File::Basename** in the file **File/Basename.pm**.

Perl searches for module files in a set of directories specified by the Perl library path. This is set when Perl is first installed. You can find out what directories Perl will search for modules in by issuing **perl -V** from the command line:

```
% perl -V
Summary of my perl5 (revision 5.0 version 6 subversion 1) configuration:
Platform:
  osname=linux, osvers=2.4.2-2smp, archname=i686-linux
...
Compiled at Oct 11 2001 11:08:37
@INC:
  /usr/lib/perl5/5.6.1/i686-linux
  /usr/lib/perl5/5.6.1
  ...
You can modify this path to search in other locations by placing the use lib command somewhere at the top of your script:
```

```
#!/usr/bin/perl

use lib '/home/lstein/lib';
use MyModule;
...
```

This tells Perl to look in **/home/lstein/lib** for the module **MyModule** before it looks in the usual places. Now you can install module files in this directory and Perl will find them.

Sometimes you really need to know where on your system a module is installed. Perldoc to the rescue again -- use the **-l** command-line option:

```
% perldoc -l File::Basename
/System/Library/Perl/5.8.8/File/Basename.pm
```

10

Making modules

Programming for Biology

11

What is a module?



```
52 sub _get_description {
53     my $service = shift;
54     my $protocol = shift;
55
56     my $description;
57     if ($protocol) {
58         $description = "($protocol";
59         if ($service) {
60             $description .= ", $service)";
61         }
62         else {
63             $description .= ")";
64         }
65     }
66     return ($description);
67 }
```

```
4 ##### The Central Dogma #####
5 #####
6 #####
7 ## Genome
8 $tRNA_seq = "ATGACCTACTAGATCATCTATGATACTCAT";
9
10 ## Transcription
11 $rRNA_seq = $tRNA_seq;
12 $rRNA_seq =~ s/T/U/gi;
13 print "$rRNA_seq\n";
14
15 ## Translation
16 $position = 0;
17 while (substr $rRNA_seq, $position, 3) {
18     $codon = substr $rRNA_seq, $position, 3;
19     print translate_codon($codon);
20     $position = $position + 3;
21 }
22 sub translate_codon {
23     if ($_[0] =~ /GC|I/) {return Ala;}
24     if ($_[0] =~ /UGC|UGU/i) {return Cys;}
```

continue

12

Module

```
package MySequence;  
# file: MySequence.pm  
use strict;  
our $EcoRI = 'ggatcc';
```

A package (or namespace) is an abstract **container** or **environment** created to hold a logical grouping of unique symbols (i.e., subroutines).

```
sub reverseq {  
    my $sequence = shift;  
    $sequence = reverse $sequence;  
    $sequence =~ tr/gatcGATC/ctagCTAG/;  
    return $sequence;  
}  
  
sub seqlen {  
    my $sequence = shift;  
    $sequence =~ s/[^\w]/g;  
    return length $sequence;  
}  
1;
```

A Perl module must end with a true value.

13

Script

```
#!/usr/bin/perl  
# file: sequence.pl  
use strict;  
use warnings;  
use MySequence;  
  
my $sequence ='gattccggatttccaaagggtcccaatttggg';  
my $complement = MySequence::reverseq($sequence);  
  
print "original = $sequence\n";  
print "complement = $complement\n";
```

Must explicitly qualify each MySequence function by using the notation

MySequence::function_name

*

14

Module using Exporter

```
package MySequence;
# file: MySequence.pm

use strict;
use base 'Exporter';

our @EXPORT = qw(reverseq seqlen);
our @EXPORT_OK = qw($EcoRI);
our $EcoRI = 'ggatcc';

sub reverseq {
    my $sequence = shift;
    $sequence = reverse $sequence;
    $sequence =~ tr/gatcGATC/ctagCTAG/;
    return $sequence;
}

sub seqlen {
    my $sequence = shift;
    $sequence =~ s/[^gatcnGATCN]//g;
    return length $sequence;
}

1;
```

*

15

Script using Exporter

```
#!/usr/bin/perl
# file: sequence.pl

use strict;
use warnings;
use MySequence;

my $sequence ='gattccggatttccaaagggttcccaatttggg';
my $complement = reverseq($sequence);

print "original = $sequence\n";
print "complement = $complement\n";
```

*

16

Exporter - Implements default import method
for modules

```
use base 'Exporter';  
  
our @EXPORT = qw(reverseq seqlen);  
our @EXPORT_OK = qw($EcoRI);
```

use base 'Exporter' Tells Perl that this module is a type of "Exporter" module

our @EXPORT = qw(reverseq seqlen) line tells Perl to export the functions **reverseq** and **seqlen** automatically.

Also, can export **qw(afunc \$scalar @array %hash)**;

our @EXPORT_OK = qw(\$EcoRI) tells Perl that it is OK for the user to import the **\$EcoRI** variable, but not to export it automatically.

17

Getopt::Long - Extended processing
of command line options

Command line operated programs traditionally take their arguments from the command line, for example filenames.

Besides arguments, these programs often take command line *options* as well. Options are not necessary for the program to work, hence the name 'option', but are used to modify its default behavior.

Example:

```
coursemain:~ dmessina$ grep -i 'AGCG' > capture.txt  
coursemain:~ dmessina$ make_fake_fasta.pl --length 100
```

18

Script using Getopt::long

```
#!/usr/bin/env perl

use strict;
use warnings;

use Getopt::Long;
my $length = 30;
my $number = 10;
my $help;
GetOptions('l|length:i' => \$length,
          'n|number:i' => \$number,
          'h|help'      => \$help);

my $usage = "make_fake_fasta.pl - generate random DNA seqs

Options:
-n <number>    the number of sequences to make (default: 10)
-l <length>     the length of each sequence      (default: 30)
";
die $usage if $help;

my @nucs = qw(A C T G);

for (my $i = 1; $i <= $number; $i++) {
    my $seq;

    for (my $j = 1; $j <= $length; $j++) {
        my $index = int(rand(4));
        my $nuc = $nucs[$index];
        $seq .= $nuc;
    }
    print ">fake$i\n";
    print $seq, "\n";
}
```

*

References & Multi-Dimensional Data Structures

Sofia Robb

What good are references?

Sometimes you need a more complex data structure than just an array or just a hash.

What if you want to keep together several related pieces of information?

Gene	Sequence	Organism
HOXB2	ATCAGCAATATACAATTATAAAGGCCTAAATTAAAA	mouse
HDAC1	GAGCGGAGCCGGGGCGGGAGGGCGGACGGAC	human

References are only addresses.

Multi-dimensional data structures are just hashes and arrays inside of hashes and arrays.

References

- References are pointers, or the address of the data
 - All data has an address in memory
 - Humans have no need to know the address
- References are useful because they are a scalar variable.
 - Arrays and hashes are not scalar variables.
 - The only kind of data that you can store in an array or hash is scalar.

We can now store hashes and arrays in hashes and arrays by storing the address!!!!

What is a reference, what do you mean by an address?

Well first, what is a variable?

A variable is a label for the location in memory of some data. This location has an address.

Scalar	<i>really means</i>	address	
\$x=1;		0x84048ec	
	SCALAR x:	<table border="1"><tr><td>1</td></tr></table>	1
1			

Array

@y = (1, 'a', 23);

really means

0x82056b4				
ARRAY y:	<table border="1"><tr><td>1</td><td>'a'</td><td>23</td></tr></table>	1	'a'	23
1	'a'	23		

How do I find you, what's your address?

A **variable** is a labeled memory address.

When we read the contents of the variable, we are reading the contents of the memory address.

0x82056b4

ARRAY y:

1	'a'	23
---	-----	----

So, what is a reference?

A **reference** is a variable that contains the memory address of some data.

It does not contain the data itself.

It contains the memory address where data is stored.

Creating a Reference

- Every time a variable is created it gets an address
- To retrieve the address or in other words, create a reference, use '\'

Creating a Reference to an Array

```
# codons for my favorite gene: HDAC
my @codons = qw(ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT TAC TAC TAC GAC GGG GAT GTT GGA AAT TAC TAT TAT);
```

```
my $address = \@codons;
print "$address\n";
```

Output:
%% ./references.pl
ARRAY(0x100812e30)

**\$address is now a
reference to the
array.**

Creating a Reference to a Hash

```
my %HDAC;  
  
$HDAC{seq}      = "MAQTQGTRRKVCYYDGDVGNYYYGQGHPMKPHRIR...";  
$HDAC{function} = "Histone Deacetylase";  
$HDAC{symbol}   = "HDAC";  
  
my $address = \%HDAC;  
print "$address\n";
```

Output:

```
%% ./references.pl  
HASH(0x10081e538)
```

Storing References

Now that we have a way to retrieve the address we can store (assign) an array or a hash in an array or hash.

- Arrays are a list of scalars
- Hashes are key/value pairs of scalars
- References are scalars

Storing a Reference as a Hash Value

```
use Data::Dumper;

my @codons = qw(ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT
TAC TAC TAC GAC GGG GAT GTT GGA AAT TAC TAT TAT);

my $codons_address = \@codons;

my %HDAC;
$HDAC{seq} = "MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...";
$HDAC{function} = "Histone Deacetylase";
$HDAC{symbol} = "HDAC";
$HDAC{codons} = $codons_address;

## using Data::Dumper to print our data structure
print Dumper \%HDAC;
```

Notice the hash reference.

output:

```
$VAR1 = {
    'symbol' => 'HDAC',
    'function' => 'Histone Deacetylase',
    'seq' => 'MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...',
    'codons' => [
        'ATG',
        'GCG',
        'CAG',
        'ACG',
        'CAG',
        'GGC',
        'ACC',
        'CGG',
        'AGG',
        'AAA',
        'GTC',
        'TGT',
        'TAC',
        'TAC',
        'TAC',
        'GAC',
        'GGG',
        'GAT',
        'GTT',
        'GGA',
        'AAT',
        'TAC',
        'TAT',
        'TAT'
    ]
};
```

Data::Dumper is a nice way to view the contents of your data structures without complicated print statements.

Or you could use the debugger.

Altering the data

Addresses/References are like Short Cuts/Aliases

- References are NOT copies of the data. They are addresses or pointers to the data
- Since a reference is like a short cut (windows) or alias (mac), when the original data changes, the change can be seen when using the reference to access the data.
- So, if @codons is changed, the hash also changes, because the hash contains only the address of the array, not a copy of the array.

Altering the Original Array affects the reference

```
my @codons = qw(ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT TAC TAC  
TAC GAC GGG GAT GTT GGA AAT TAC TAT TAT);  
  
my $codons_address = \@codons;  
  
my %HDAC;  
$HDAC{seq} = "MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...";  
$HDAC{function} = "Histone Deacetylase";  
$HDAC{symbol} = "HDAC";  
$HDAC{codons} = $codons_address;  
  
#Replacing the contents of @codons with only 2 codons  
@codons = qw(ATG GCG);  
  
#Printing the unaltered %HDAC  
print Dumper \%HDAC;
```

Output:

```
$VAR1 = {  
    'symbol' => 'HDAC',  
    'function' => 'Histone Deacetylase',  
    'seq' => 'MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...',  
    'codons' => [  
        'ATG',  
        'GCG'  
    ]  
};
```

Only @codons was altered but the hash also changed

Anonymous Data structures

- You do not always need to retrieve the address of data to store/assign in a variable.
- You can create an anonymous array or hash on the fly.
 - It is anonymous because it is unnamed.
 - It only has an address, no name, no label.
- We use the [] in the anonymous array assignment
- We use the {} in the anonymous hash assignment.

Creating and Storing an Anonymous Array

the array is never given a name.

Before:

```
my @codons = qw(ATG GCG) ;  
my $address = \@codons;  
$HDAC{codons} = $address;
```

Now: Creating and storing an anonymous array
`$HDAC{codons} = ["ATG" , "GCG"] ;`



Notice the [] instead of ().

Storing an Anonymous (unnamed) Array as a Hash Value

```
#my @codons = qw(ATG GCG);  
  
my %HDAC;  
$HDAC{seq} = "MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...";  
$HDAC{function} = "Histone Deacetylase";  
$HDAC{symbol} = "HDAC";  
$HDAC{codons} = [ "ATG" , "GCG" ] ;           the array is never given a name.  
  
print Dumper \%HDAC;
```

Output:

```
$VAR1 = {  
    'symbol' => 'HDAC',  
    'function' => 'Histone Deacetylase',  
    'seq' => 'MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...',  
    'codons' => [  
        'ATG',  
        'GCG'  
    ]  
};
```

Storing an Anonymous (unnamed) Hash as a Hash Value

```
my %HDAC;  
$HDAC{seq} = "MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...";  
$HDAC{function} = "Histone Deacetylase";  
$HDAC{symbol} = "HDAC";  
$HDAC{codons} = [ "ATG" , "GCG" ] ;  
$HDAC{expression} = { "liver" => 2.1 , "heart" => 1.3 } ;  
  
print Dumper \%HDAC;
```



Notice the {} instead of ().

Output:

```
$VAR1 = {  
    'symbol' => 'HDAC',  
    'function' => 'Histone Deacetylase',  
    'expression' => {  
        'heart' => '1.3',  
        'liver' => '2.1'  
    },  
    'seq' => 'MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...',  
    'codons' => [  
        'ATG',  
        'GCG'  
    ]  
};
```

Storing an Anonymous (unnamed) Hash as a Hash Value

```
$HDAC{expression} = { "liver" => 2.1 ,  
                      "heart" => 1.3  
                    } ;
```

Same As:

```
$HDAC{expression}{"liver"} = 2.1 ;  
$HDAC{expression} {"heart"} = 1.3 ;
```

Looks Like:

```
$HDAC{symbol}      = "HDAC";
```

Now, all the data is in the data structure, how to you get it out?

Whole chunks of data or pieces of data can be retrieved from the multidimensional structures by using the address.

A.K.A. **Dereferencing**

3 Easy Steps to Dereference

Dereference === retrieve data from address

1. Get the address, or reference: \$ADDRESS
2. Wrap the address, or reference in {}: {\$ADDRESS}
3. Put the symbol of the data type out front @: @{\$ADDRESS}

Dereference a reference to an array

```
my @codons = qw(ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT TAC  
TAC TAC GAC GGG GAT GTT GGA AAT TAC TAT TAT);  
  
my $codons_address = \@codons;  
  
print "address of the array:\n$codons_address\n\n";  
print "array from a dereferenced reference:\n @{$codons_address}\n";
```

Output:

```
address of the array:  
ARRAY(0x7fd89c016b90)  
  
array from a dereferenced reference:  
ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT TAC TAC GAC GGG GAT GTT GGA  
AAT TAC TAT TAT
```

Dereference an anonymous array that is a hash value

```
Key           Value
$HDAC{codons} = [ "ATG" , "GCG" ] ;  
  
my $codons_address = $HDAC{codons};  
  
print "address of the array:\n$codons_address\n\n";  
print "array from a dereferenced reference:\n @{$codons_address}\n";
```

Output:

address of the array:
ARRAY(0x7f97db822958)

array from a dereferenced reference:
ATG GCG

Regular hash

```
$hash{key} = "value";  
my $value = $hash{key};
```

Did you notice that dereferencing an array and an anonymous array are the same?

Dereference an anonymous hash that is a hash value

```
$HDAC{expression} = { "liver" => 2.1 , "heart" => 1.3 } ;  
  
my $hash_address = $HDAC{expression};  
  
print "address of the hash:\n$hash_address\n\n";  
print "keys from a dereferenced reference:\n\n";  
  
my @keys = keys %{$hash_address};  
  
print "keys from a dereferenced reference:\n@keys\n";
```

Output:

address of the hash:
HASH(0x7f94e38226d0)

keys from a dereferenced reference:
heart liver

Regular hash

```
my @keys = keys %hash;
```

It is not always needed to explicitly retrieve the address

```
$HDAC{expression} = { "liver" => 2.1 , "heart" => 1.3 } ;  
  
my $hash_address = $HDAC{expression};  
my @keys = keys %{$hash_address};  
  
my @keys = keys %{ $HDAC{expression} };  
This evaluates to an address  
  
print "keys from a dereferenced reference:\n@keys\n";
```

Output:

```
keys from a dereferenced reference:  
heart liver
```

Regular hash

```
my @keys = keys %hash;
```

Dereferencing to access every element from the anonymous array that is a hash value

```
$HDAC{codons} = [ "ATG" , "GCG" ] ;  
  
##my @codons = @{$HDAC{codons}} ;  
##my $zeroth_element = ${$HDAC{codons}}[0];  
  
foreach my $codon ( @ { $HDAC{codons} } ) {  
    This evaluates to  
    an address  
  
    print "codon: $codon\n";  
}
```

Output:

```
codon: ATG  
codon: GCG
```

Regular array:

```
foreach my $codon ( @codons )  
{  
    print "codon: $codon\n";  
}
```

Dereferencing to access a piece of the anonymous array that is a hash value.

```
$HDAC{codons} = [ "ATG" , "GCG" ] ;  
##my @codons = @{$HDAC{codons}} ;  
  
my $zeroth_element = ${$HDAC{codons}}[0];  
This evaluates to  
an address  
  
print "the 0th element = $zeroth_element\n";
```

Output:

the 0th element = ATG

Regular array

```
$array[1] = "value";  
my $value = $array[1]
```

Dereferencing to access every key/value pair from the anonymous hash in a hash

```
$HDAC{expression} = { "liver" => 2.1 , "heart" => 1.3 } ;  
  
foreach my $tissue ( keys %{$HDAC{expression}} ) {  
    my $level = ${$HDAC{expression}}{$tissue};  
    print "$tissue: $level\n";  
}
```

Output:

heart: 1.3
liver: 2.1

Regular Hash

```
foreach my $key (keys %hash) {  
    my $value = $hash{$key};  
}
```

The `ref()` function

`ref(REF)`

returns the data type in which the reference points

```
my %hash;
```

```
$hash{codons} = [ "ATG" , "TTT" ] ;
```

```
my $address = $hash{codons} ;
```

```
ref ( $address ) ;          ## returns ARRAY
```

```
ref ( $hash{codons} ) ;     ## returns ARRAY
```

both `$address` and `$hash{codons}` evaluate to the address of the array

Extra fun stuff to look over later.

- Array of arrays
- Another Scripting Example:
 - Creating a Hash of Hashes

Multidimensional Data: Making an Array of Arrays

```
my @spotarray = (
    [0.124, 43.2, 0.102, 80.4],
    [0.113, 60.7, 0.091, 22.6],
    [0.084, 112.2, 0.144, 35.3]
);

# two ways to get the value of the inner index
# my $cell_1_0 = ${$spotarray[1]}[0];
my $cell_1_0 = $spotarray[1][0];

print $cell_1_0;
```

Output:

0.113

Scripting Example: Creating a Hash of Hashes

We are presented with a table of sequences in the following format: the ID of the sequence, followed by a tab, followed by the sequence itself.

```
2L52.1      atgtcaatggtaagaaatgtatcaaatcagagcgaaaaattggaagttaag...
4R79.2      tcaaatacagcaccagctccttttatagttcgaaattaatgtccaact...
AC3.1       atggctcaaactttactatcacgtcattccgtggtgtcaactgttatt...
...
```

For each sequence calculate the length of the sequence and the count for each nucleotide. Store the results into hash of hashes in which the outer hash's key is the ID of the sequence, and the inner hashes' keys are the names and counts of each nucleotide.

```

#!/usr/bin/perl -w

use strict;

# tabulate nucleotide counts, store into %sequences

my %seqs; # initialize hash
while (my $line = <>) {
    chomp $line;
    my ($id,$sequence) = split "\t",$line;
    my @nucleotides = split '', $sequence; # array of base pairs
    foreach my $n (@nucleotides) {
        $seqs{$id}{$n}++; # count nucleotides and keep tally
    }
}

# print table of results
print join("\t",'id','a','c','g','t'),"\n";

foreach my $id (sort keys %seqs) {
    print join("\t",$id,
               $seqs{$id}{a},
               $seqs{$id}{c},
               $seqs{$id}{g},
               $seqs{$id}{t}),
               ),"\n";
}

```

The output will look something like this:

id	a	c	g	t
2L52.1	23	4	12	11
4R79.2	15	12	5	18
AC3.1	11	11	8	20
...				

Object Oriented Programming and Perl

Prog for Biol 2011
Simon Prochnik

Sunday, October 21, 12

1

Why do we teach you about objects and object-oriented programming (OOP)?

- Objects and OOP allow you to use other people's code to do a lot in just a few lines.
- For example, in the lecture on bioperl, you will see how to search GenBank by a sequence Accession, parse the results and reformat the sequence into any format you need in less than a dozen lines of object-oriented perl. Imagine how long it would take to write that code yourself!
- Someone else has already written and tested the code, so you don't have to.
- Most people don't ever write an object of their own: only create your own modules and objects if you have to
- search CPAN (www.cpan.org) to see if there is already a module that does what you need. There were 18,534 modules on Oct 14th 2010, this has grown to 100,575 (Oct 20, 2011), 114,367 Oct 19, 2012! Surely you can find a module to do what you want.

Sunday, October 21, 12

2

Using objects in perl

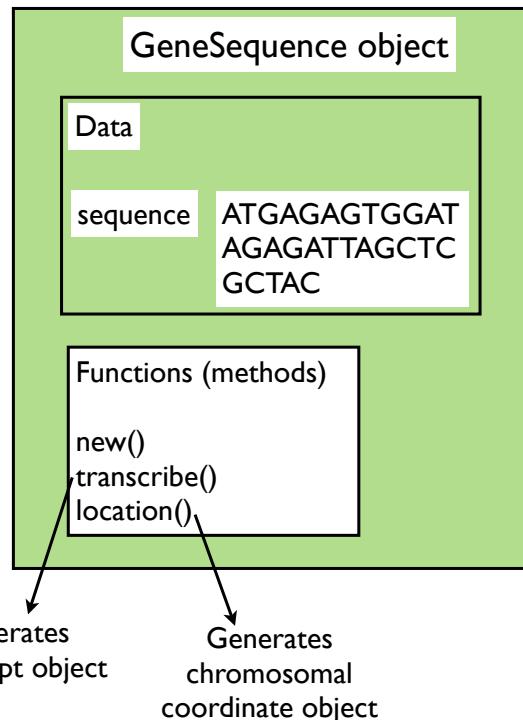
- some examples to show how you can use objects

Sunday, October 21, 12

3

Object-oriented programming is a programming style

- An object is a special kind of data structure (variable) that stores specific kinds of data and automatically comes with functions (methods) that can do useful things with that data
- Objects are often designed to work with data and functions that you would find associated with a real-world object or thing, for example, we might design gene sequence objects.
- A gene sequence object might store its chromosomal position and sequence data and have functions like `transcribe()` and `new()` to create a new object.



Sunday, October 21, 12

4

An example of a Microarray object that is designed specifically to handle microarray data

```
#!/usr/bin/perl
#File: 00_script.pl
use strict;
use warnings;
use Microarray; # I wrote this example object class
my $microarray = Microarray->new( gene => 'CDC2',
                                    expression => 45,
                                    tissue => 'liver',
                                    );
my $gene_name = $microarray->gene();
print "Gene for this microarray is $gene_name\n";
my $tissue = $microarray->tissue();
print "The tissue is $tissue\n";
```

Tell perl you want to use objects in the Microarray class

Create a new object and load data

call the gene() subroutine to get gene name data from the object

call the tissue() subroutine to get tissue data from the object

Output on screen:
Gene for this microarray is CDC2
The tissue is liver

Sunday, October 21, 12

5

An example that deals with statistics (Statistics::Descriptive objects)

```
#!/usr/bin/perl
#File: mean_and_variance.pl
use strict;
use warnings;

use Statistics::Descriptive; # this is on cpan.org
# need to make new object with S::D::Full->new()
my $stat = Statistics::Descriptive::Full->new();
$stat->add_data(1,2,3,4);
my $mean = $stat->mean();
my $var = $stat->variance();
print "mean is $mean\n";
print "variance is $variance\n";
```

Make new object with new()

Add data

Calculate mean

Calculate variance

Output on screen:
mean is 2.5
variance is 1.66666666666667

Sunday, October 21, 12

6

An example that deals with statistics (Statistics::Descriptive objects)

Make new object
with new()

Add data

Calculate mean

Calculate variance

```
#!/usr/bin/perl
#File: mean_and_variance.pl
use strict;
use warnings;

use Statistics::Descriptive;

my $stat = Statistics::Descriptive->new();
$stat->add_data(1,2,3,4);
my $mean = $stat->mean();
my $var  = $stat->variance();
print "mean is $mean\n";
print "variance is $variance\n";
```

Output on screen:
mean is 2.5
variance is 1.66666666666667

Sunday, October 21, 12

7

Let's look at the new OOP syntax in more detail

```
# tell perl you want to use objects
# in a certain class
use Statistics::Descriptive;
```

Here's the class name
'Statistics::Descriptive'. perl will look for a
module with the filename
..../Statistics/Descriptive.pm

Sunday, October 21, 12

8

Let's look at the new OOP syntax in more detail

Before you can use an object, you create one.
This is often done with a call to a new() method.

```
#create a new object in the  
# 'Statistics::Descriptive' class  
my $stat = Statistics::Descriptive->new();
```

An object in perl is a scalar variable (a special one that belongs to a Class). All scalar variables start with \$

We tell perl which Class of object we want to create

This arrow -> goes between the class and the new method

new() creates a new object. Every Class has a new() method

Sunday, October 21, 12

9

Let's look at the new OOP syntax in more detail

Once you have created an object you call methods on it to use the object

```
# call a method (subroutine) on the $stat  
# object  
$stat->add_data(1,2,3,4);
```

Here's the object

This arrow -> goes between the object and the method (subroutine) name

add_data is the name of the method we are calling. A method is just a subroutine

The data we are passing in is the numbers 1,2,3 and 4. These numbers are being passed into the subroutine add_data()

Sunday, October 21, 12

10

Object-oriented programming in a little more detail

- Let's look at which elements of perl are used to provide object oriented programming

Sunday, October 21, 12

11

Object Oriented Programming and Perl

- To understand object-oriented syntax in perl, we need to recap three things: **references, subroutines, packages**.
- These three elements of perl are recycled with slightly different uses to provide object-oriented programming

What you can do	Normal perl (procedural perl)	Object-oriented perl
organize code that goes together for reuse	package	class (the type or kind of object, and all the code that goes with it)
store data (simple or very complex)	a reference	the object itself (a reference to a data structure)
work on data by writing simple code	subroutine	a method (function that acts on the object)

Sunday, October 21, 12

12

Object Oriented Programming and Perl

- The OOP paradigm provides i) a solid framework for sharing code -- reuse
- and ii) a guarantee or contract or specification for how the code will work and how it can be used -- an interface
- and iii) hides the details of implementation so you only have to know how to use the code, not how it works -- saves you time, quick to learn, harder to introduce bugs
- Here we are briefly introducing you to OOP and objects so that you can quickly add code that's already written into your scripts, rather than spend hours re-inventing wheels. Many more people use objects than write them.

Sunday, October 21, 12

13

I: Recap references

example of syntax

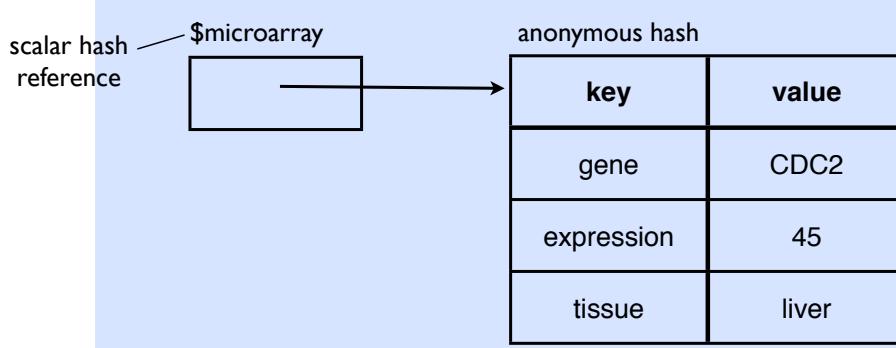
```
$ref_to_hash = {key1=>'value1',key2=>'value2',...}
```

code example

```
my $microarray = {gene => 'CDC2',
                  expression => 45,
                  tissue => 'liver',
                };
```

We can store any pieces of data we would like to keep together in a hash

Here is the data structure in memory



Sunday, October 21, 12

14

II: recap subroutines

- solve a problem, write code once, and call the code simply
- reusing a single piece of code instead of copying, pasting and modifying reduces the chance you'll make an error and simplifies bug fixing.

```
#!/usr/bin/perl -w
use strict;
my $seq;
while (my $seqline = <>) { # read sequence from standard in
    my $clean    = cleanup_sequence($seqline); # clean it up
    $seq        .= $clean;                      # add it to full sequence
}
sub cleanup_sequence {
    my ($sequence) = @_;
    $sequence = lc $sequence; # translate everything into lower case
    $sequence =~ s/[\s\d]//g; # remove whitespace and numbers
    $sequence =~ m/^+[gatcn]+$/ or die "Sequence contains invalid
                                         characters!";
    return $sequence;
}
```

Sunday, October 21, 12

15

III: now let's recap packages

- organise code that goes together into reusable modules, packages

```
#!/usr/bin/perl -w                                     read_clean_sequence.pl
#File: read_clean_sequence.pl
use strict;
use Sequence;
my $seq;
while (my $seqline = <>) { # read sequence from standard in
    my $clean    = cleanup_sequence($seqline); # clean it up
    $seq        .= $clean;                      # add it to full sequence
}

#file: Sequence.pm                                     Sequence.pm
package Sequence;
use strict;
use base Exporter;
our @EXPORT = ('cleanup_sequence');
sub cleanup_sequence {
    my ($sequence) = @_;
    $sequence = lc $sequence; # translate everything into lower case
    $sequence =~ s/[\s\d]//g; # remove whitespace and numbers
    $sequence =~ m/^+[gatcn]+$/ or die "Sequence contains invalid
                                         characters!";
    return $sequence;
}
1;
```

Sunday, October 21, 12

16

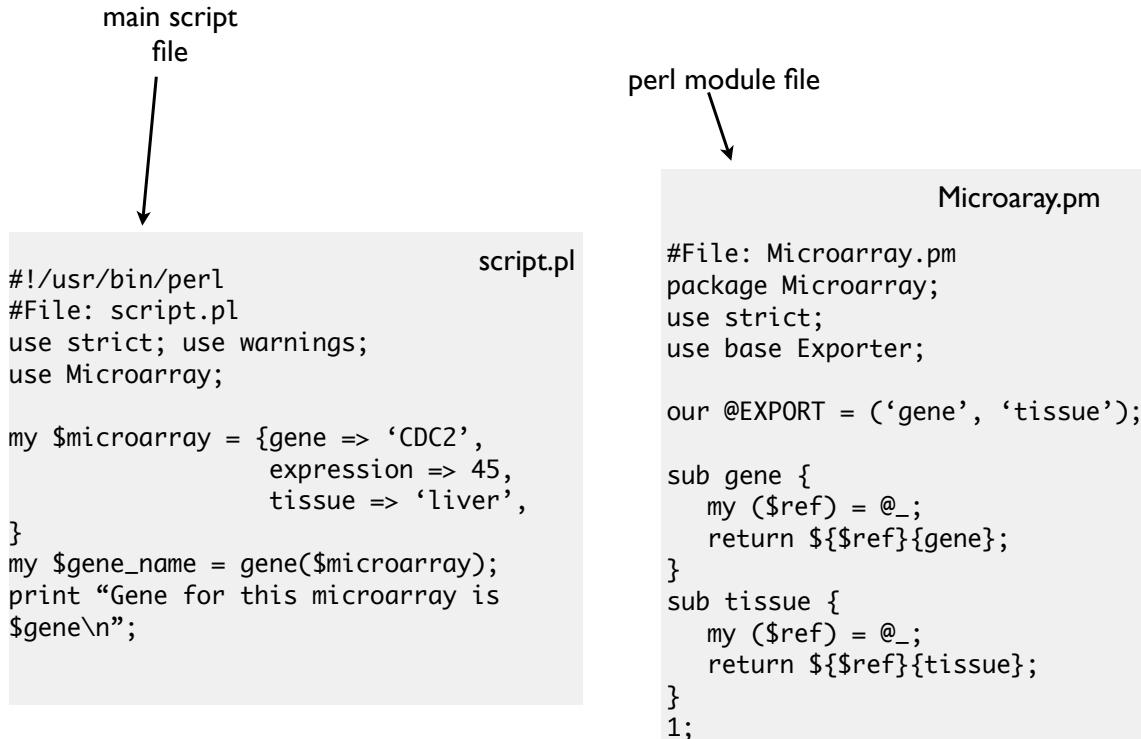
Let's recap subroutines: new example with references

```
#!/usr/bin/perl
use strict;
use warnings;
my $microarray = { gene => 'CDC2',
                  expression => 45,
                  tissue => 'liver',
                };
...
my $gene_name = gene($microarray);
...
sub gene {
    my ($ref) = @_;
    return ${$ref}{gene};
}
sub tissue {
    my ($ref) = @_;
    return ${$ref}{tissue};
}
```

Sunday, October 21, 12

17

recap packages



Sunday, October 21, 12

18

Let's look at how you create object code

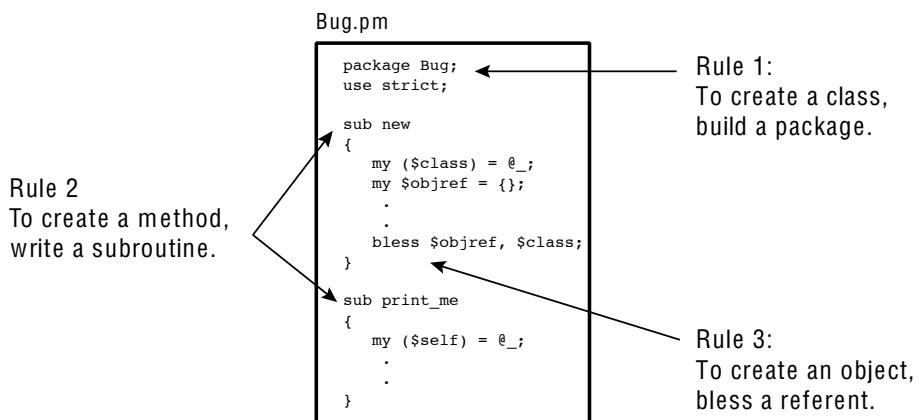
- This is mostly for reference.
- You'll probably use it rarely, if at all

Sunday, October 21, 12

19

Three Little Rules

- Rule 1: To create a class, build a package
- Rule 2: To create a method, write a subroutine
- Rule 3: To create an object, bless a reference



Sunday, October 21, 12

20

Rule 1: To create a class, build a package

- all the code that goes with an object (methods, special variables) goes inside a special package
 - perl packages are just files whose names end with '.pm' e.g. Microarray.pm
 - package filenames should start with a capital letter
 - the name of the perl package tells us the class of the object. This is really the type or kind of object we are dealing with.
- Microarray.pm is a package, so it will be easy to convert into object-oriented code

Sunday, October 21, 12

21

Rule 2: To create a method, write a subroutine

- we already have `gene()` in Microarray.pm
- this can be turned into a method
- we need one extra subroutine to create new objects
- the creator method is called `new()` and has one piece of magic...

Sunday, October 21, 12

22

Rule 3: To create an object, bless a reference

- The new() subroutine uses the bless function to create an object
- full details coming up... but here's the skeleton of a new() method

```
sub new {  
    ...  
    my $self = {};  
    bless $self, $class;  
    ...  
}
```

create a reference, a
hashref {} is the most
common seen in perl

bless a reference
into a class

Sunday, October 21, 12

23

Let's recap packages

```
#!/usr/bin/perl -w  
#File: script.pl  
use strict;  
use Microarray;  
  
my $microarray = { gene => 'CDC2',  
                  expression => 45,  
                  tissue => 'liver',  
                };  
my $gene_name = gene($microarray);  
print "Gene for this microarray is $gene\n";
```

```
#File: Microarray.pm  
package Microarray;  
use strict;  
use base Exporter;  
  
our @EXPORT = ('gene', 'tissue');  
  
sub gene {  
    my $ref = shift;  
    return ${$ref}{gene};  
}  
sub tissue {  
    my $ref = shift;  
    return ${$ref}{tissue};  
}  
1;
```

Sunday, October 21, 12

24

Transforming a package into an object-oriented module or class

procedural perl package
(what you saw yesterday)

...transforming the package into a class...

```
#File: Microarray.pm
package Microarray;
use strict;
use base Exporter;

our @EXPORT = ('gene', 'tissue');

sub gene {
    my ($ref) = @_;
    return ${$ref}{gene};
}

sub tissue {
    my ($ref) = @_;
    return ${$ref}{tissue};
}

1;
```



```
#File: Microarray.pm
package Microarray;
use strict;

sub gene {
    my $self = shift; # same as my ($self) = @_;
    return ${$self}{gene};
}

sub tissue {
    my $self = shift;
    return ${$self}{tissue};
}

1;
```

Sunday, October 21, 12

25

The new() method is a subroutine that creates a new object

```
sub new {
    my $class = shift;
    my %args = @_;
    my $self = {};
    foreach my $key (keys %args) {
        ${$self}{$key} =
            $args{$key};
    }
    # the magic happens here
    bless $self, $class;
    return $self;
}
```

the first argument is always the class of the object you are making. **perl gives you this as the first argument automatically**

a hash reference is the data structure you build an object from in perl

here we initialize variables in the object (in case there are any)
Some people like to write
`${$self}{$key}`
as
`$self -> {$key}`

bless makes the object \$self (which is a hash reference) become a member of the class \$class

Sunday, October 21, 12

26

bless creates an object by making a reference belong to a class

Make an anonymous hash in the debugger

```
$a = {};
p ref $a;
HASH
```

Make a MySequence object in the debugger

```
$self = {};
$class = 'MySequence';
bless $self, $class;

x $self
0  MySequence=HASH(0x18bd7cc)
    empty hash
p ref $a
MySequence
```

Sunday, October 21, 12

27

final step

object-oriented module or class

```
#File: Microarray.pm
package Microarray;
use strict;

sub new {
    my $class = shift;
    my %args = @_;
    my $self = {};
    foreach my $key (keys %args) {
        ${$self}{$key} = $args{$key};
    }
    # the magic happens here
    bless $self, $class;
    return $self;
}

sub gene {
    my $self = shift;
    return ${$self}{gene};
}
sub tissue {
    my $self = shift;
    return ${$self}{tissue};
}
1;
```

Sunday, October 21, 12

28

OOP script

```
#!/usr/bin/perl
use strict; use warnings;      procedural version
#File: script.pl
my $microarray = { gene => 'CDC2',
                  expression => 45,
                  tissue => 'liver',
                };
my $gene_name = gene($microarray);
print "Gene for this microarray is $gene\n";
```

```
#!/usr/bin/perl
#File: OO_script.pl                                     OO version
use strict; use warnings;
use Microarray;
my $microarray = Microarray->new( gene => 'CDC2',
                                    expression => 45,
                                    tissue => 'liver',
                                  );
my $gene_name = $microarray->gene();
print "Gene for this microarray is $gene_name\n";
my $tissue = $microarray->tissue();
print "The tissue is $tissue\n";
```

Sunday, October 21, 12

29

Lastly, did I mention “code lazy”?

- This lecture has introduced you to object-oriented programming
- You only need to **use** other people’s objects (beg, borrow, buy, steal).
- Only create your own modules and objects if you **have to**.

Sunday, October 21, 12

30

Problems

1. Take a look at the Statistics::Descriptive module on cpan here <http://search.cpan.org/~shlomif/Statistics-Descriptive-3.0202/lib/Statistics/Descriptive.pm>

2. Write a script that uses the methods in Statistics::Descriptive to calculate the standard deviation, median, min and max of the following numbers

12,-13,-12,7,11,-4,-12,9,6,7,-9

Optional questions

4. Add a method to Microarray.pm called expression() which returns the expression value

5. Currently calling \$a = \$m->gene() gets the value of gene in the object \$m. Modify the gene() method so that if you call gene() with an argument, it will set the value of gene to be that argument e.g.

```
$m->gene('FOXP1');           # this should set the  
                             # gene name to 'FOXP1'  
print $m->gene();    # this should print the value 'FOXP1'
```

Further reading on inheritance

- If you want to make an object that is a special case or subclass of another, more general, object, you can have it inherit all the general data storage and functions of the more general object.
- This saves coding time by re-using existing code. This also avoids copying and pasting existing code into the new object, a process that makes code harder to maintain and debug.
- For example, a MicroRNA_gene object is a special case of a Gene object and might have some specific functions like cut_RNA_hairpin() as well as general functions like transcribe() it can **inherit** from the general gene object.
- More formally, a subclass inherits variables and functions from its superclass (like a child and a parent). Here are some examples

```
package MicroRNA;  
use base 'Gene'; # Gene is a parent  
use base 'Exporter'; # Exporter is another parent
```

Good Practices for Writing Perl Pipelines

Using perl as bioinformatics glue

Simon Prochnik
with code from Scott Cain

Sunday, October 21, 12

1

Built-in perldoc <perl topic> to get help

```
% perldoc perlref
PERLREF(1)          User Contributed Perl Documentation      PERLREF(1)

NAME
    perlref - Perl references and nested data structures

NOTE
    This is complete documentation about all aspects of references. For a
    shorter, tutorial introduction to just the essential features, see
    perlreftut.

DESCRIPTION
    Before release 5 of Perl it was difficult to represent complex data
    structures, because all references had to be symbolic--and even then it
    was difficult to refer to a variable instead of a symbol table entry.
    Perl now not only makes it easier to use symbolic references to
    ...

```

Also available online at <http://perldoc.perl.org/index-tutorials.html>

Sunday, October 21, 12

2

Built-in perldoc -f <command> to get help

```
% perldoc -f split

split /PATTERN/,EXPR,LIMIT
split /PATTERN/,EXPR
split /PATTERN/
split    Splits the string EXPR into a list of strings and returns that
         list. By default, empty leading fields are preserved, and
         empty trailing ones are deleted. (If all fields are empty,
         they are considered to be trailing.)
```

Sunday, October 21, 12

3

Get online help from perldoc.perl.org

<http://perldoc.perl.org/functions/split.html>

The screenshot shows the perldoc.perl.org website. The top navigation bar has a logo of an onion, the text 'perldoc.perl.org', and 'Perl Programming Documentation'. On the right, there's a search icon and a link to 'Show recent page'. The main content area shows the 'split' function documentation under 'Functions > split'. The left sidebar has sections for 'Perl version' (with a dropdown menu), 'Manual' (with links to Overview, Tutorials, FAQs, History / Changes, and License), and 'Reference' (with links to Language, Functions, Operators, Special Variables, Pragmas, Utilities, Internals, and Platform Specific). The right sidebar includes links to 'Perl functions A-Z', 'Perl functions by category', and 'The 'perifunc' manpage'. The main content area contains the Perl documentation for the split function, which defines it as splitting a string into a list of strings based on a pattern, with examples and notes about empty fields and whitespace.

Sunday, October 21, 12

4

Running your script in the perl debugger

```
> perl -d myScript.pl
Loading DB routines from perl5db.pl version 1.28
Editor support available.
Enter h or `h h' for help, or `man perldebug' for more help.
main:::(myScript.pl:3): print "hello world\n";
DB<1>

h          help
q          quit
n or s    next line or step through next line
<return>  repeat last n or s
!          repeat last command
c 45      continue to line 45
b 45      break at line 45
b 45 $a == 0 break at line 45 if $a equals 0
p $a      print the value of $a
x $a      unpack or extract the data structure in $a
R          restart the script
```

Sunday, October 21, 12

5

The interactive perl debugger

```
> perl -de 4
Loading DB routines from perl5db.pl version 1.28
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main:::(-e:1):4
DB<1> $a = {foo => [1,2] , boo => [2,3] , moo => [6,7]}
DB<2> x $a
0 HASH(0x8cd314)
  'boo' => ARRAY(0x8c3298)
    0 2
    1 3
  'foo' => ARRAY(0x8d10d4)
    0 1
    1 2
  'moo' => ARRAY(0x815a88)
    0 6
    1 7
```

Sunday, October 21, 12

6

More perl tricks: one line perl

```
> perl -e <COMMAND>

> perl -e '@a = (1,2,3,4);print join("\t",@a),"\n"'
1      2      3      4

#print IDs from fasta file
> perl -ne 'if (/^>(\S+)/) {print "$1\n"}' volvox_AP2ERE吕布.fa
vca4886446_93762
vca4887371_120236
vca4887497_89954
                                         Contents of fasta file volvox_AP2ERE吕布.fa

• see Chapter 19, p.
  492-502 Perl book 3rd ed.

>vca4886446_93762
MSPPPHTSTTESRMAPPQSSTPSGDVDGS
>vca4887371_120236
MAGLHSVPKLSARRPDWELPELHSDLQLAP
>vca4887497_89954
MAYKLFGTAAVLNYDLPAERRAELDAMSME
>vca4888938_93984
MLHTDLQPPRCRTSGPRPDPLRMETRARER
```

Sunday, October 21, 12

7

Is a module installed?

one-line perl program with '-e'

this is the program in quotes

% perl -e 'use Bio::AlignIO::clustalw' ← all ok: no errors

The module in the next example hasn't been installed
(it doesn't actually exist)

% perl -e 'use Bio::AlignIO::myformat'
Can't locate Bio/AlignIO/myformat.pm in
@INC (@INC contains: /sw/lib/perl5 /sw/
lib/perl5/darwin /Users/simongp/lib /
Users/simongp/Library/Perl/5.8.1/darwin-
thread-multi-2level /Users/simongp/
Library/Perl/5.8.1 /Users/simongp/
com_lib /Users/simongp/cvs/bdgp/software/
perl-modules ...

To install a module

% sudo cpan
install Bio::AlignIO::clustalw

perl can't find the module in any of
the paths in the PERL5LIB list (which
is in the perl variable @INC)
You can add directories with
use lib '/Users/yourname/lib';
after the use strict; at the beginning
of your script

Sunday, October 21, 12

8

Toy example: Finding out how to run a small task

- Let's assume we have a multiple fasta file and we want to use perl to run the program clustalw to make a multiple sequence alignment and read in the results.
- Here are some sequences in fasta format

```
>vca4886446_93762
MSPPPTHSTTESRMAPPQSSTPSGDVDGS
>vca4887371_120236
MAGLHSVPKLSARRPDWELPELHGDLQLAP
>vca4887497_89954
MAYKLFGTAAVLNYDLPAERRAELDAMSME
>vca4888938_93984
MLHTDLQPPRCRTSGPRPDPLRMETRARER
```

Here is the pipeline:

get fasta seq filename,
construct output filename,
generate command line that will align sequences with clustalw,
read in/parse output file,
(do something with the data)

How do we start on this? -- Looking for help

- Google

- <program name> documentation / docs / command line
- eg google 'clustal command line'

USE OF OPTIONS

All parameters of Clustalw can be used as options with a "-" That permits to use Clustalw in a script or in batch.

```
$ clustalw -options
CLUSTAL W (1.7) Multiple Sequence Alignments
clustalw option list:-
    -help
        -options
        -infile=filename
        -outfile=filename
        -type=protein OR dna
        -output=gcg OR gde OR pir OR phylip
```

Build a command line from the options you need **and test it out**

USE OF OPTIONS

All parameters of Clustalw can be used as options with a "-" That permits to use Clustalw in a script or in batch.

```
$ clustalw -options
CLUSTAL W (1.7) Multiple Sequence Alignments
clustalw option list:-
    -help
        -options
        -infile=filename
        -outfile=filename
        -type=protein OR dna
        -output=gcg OR gde OR pir OR phylip
```

Command line would be:

```
% clustalw -infile=ExDNA.fasta -outfile=ExDNA.aln -type=dna
Did it do exactly what you want/expect when you tested it?
```

Sunday, October 21, 12

11

Running a command line from perl

Command line

```
clustalw -infile=ExDNA.fasta -outfile=ExDNA.aln -type=dna
```

Script

```
#!/usr/bin/perl
use strict; use warnings;

my $file = 'ExDNA.fasta';
my $clustFile = 'ExDNA.aln';
# build command line
my $cmd = "clustalw -infile=$file -outfile=$clustFile -type=dna";
print "Call to clustalw $cmd\n";      # show command
my $oops = system $cmd;      # system call and save return
                                # value in $oops
die "FAILED $" if $oops;      # $oops true if failed
```

Sunday, October 21, 12

12

Util.pm package for nice reusable utility functions

```
package Util;
use strict;
our @EXPORT = qw(do_or_die);      # allow do_or_die() to be exported
                                    # without specifying
                                    # Util::do_or_die()
use Exporter;
use base 'Exporter';

# -----
sub do_or_die {
    my $cmd = shift;
    print "CMD: $cmd\n";
    my $oops = system $cmd;
    die "Failed" if $oops;
}
# -----

1;
```

Sunday, October 21, 12

13

Util.pm in a script

```
#!/usr/bin/perl
use strict; use warnings;
use lib 'lib'; # you might need to tell perl where to find
Util.pm
                    # or with something like this
                    # use lib '/Users/simongp/lib';
use Util;

my $file = 'ExDNA.fasta';
my $clustFile = 'ExDNA.aln';
my $cmd = "clustalw -infile=$file -outfile=$clustFile
           -type=dna";           # build command line
#print "Call to clustalw $cmd\n";      # don't need this
#  anymore because do_or_die shows the command

do_or_die($cmd);      # I use this all the time
```

Sunday, October 21, 12

14

Next step: How do we find out how to parse the clustalw alignment file (without even knowing what the file format is)?

The output is a clustalw multiple sequence alignment in the file ExDNA.aln

Look in bioperl documentation for help.

See HOWTOs

<http://www.bioperl.org/wiki/HOWTOs>

BioPerl HOWTOs

Beginners HOWTO

An introduction to BioPerl, including reading and writing sequence files, running and parsing BLAST, retrieving from databases, and more.

SeqIO HOWTO

Sequence file I/O, with many script examples.

...

AlignIO and SimpleAlign HOWTO

A guide on how to create and analyze alignments using BioPerl.

Help on AlignIO from bioperl

Abstract

This is a HOWTO that talks about using AlignIO and SimpleAlign to create and analyze alignments. It also discusses how to run various applications that create alignment files.

AlignIO

Data files storing multiple sequence alignments appear in varied formats and Bio::AlignIO is the Bioperl object for conversion of alignment files. AlignIO is patterned on the Bio::SeqIO object and its commands have many of the same names as the commands in SeqIO. Just as in SeqIO the AlignIO object can be created with "-file" and "-format" options:

```
use Bio::AlignIO;
my $io = Bio::AlignIO->new(-file  => "receptors.aln",
                           -format => "clustalw");
```

If the "-format" argument isn't used then Bioperl will try and determine the format based on the file's suffix, in a case-insensitive manner. Here is the current set of input formats:

Format	Suffixes	Comment
bl2seq		
clustalw	.aln	

More help on AlignIO from bioperl

Here's a more useful synopsis

```
use Bio::AlignIO;
$in = Bio::AlignIO->new(-file => "inputfilename",
                         -format => 'fasta');
$out = Bio::AlignIO->new(-file => ">outputfilename",
                         -format => 'pfam');

while ( my $aln = $in->next_aln ) {
    $out->write_aln($aln);
}
```

Let's add this to our script

Sunday, October 21, 12

17

Use bioperl to parse the clustalw alignment

Command line

```
clustalw -infile=ExDNA.fasta -outfile=ExDNA.aln -type=dna
```

Script

```
#!/usr/bin/perl
use strict; use warnings;
use Util;
use Bio::AlignIO;
my $file = 'ExDNA.fasta';
my $clustFile = 'ExDNA.aln';
my $cmd = "clustalw -infile=$file -outfile=$clustFile
           -type=dna";                      # build command line
do_or_die($cmd);
my $in = Bio::AlignIO->new(-file    => $clustFile,
                           -format => 'clustalw');
while ( my $aln = $in->next_aln() ) {
    ...
}
```

Sunday, October 21, 12

18

We just wrote a script to parse in a clustalw alignment without having to worry about the file format

- That's the point of bioperl and object-oriented programming.
- You don't need to know the details of the file format to be able to work with it or how the alignment is stored in memory.
- Here's a sample file in case you are curious

```
CLUSTAL W (1.74) multiple sequence alignment

seq1 -----KSKERYKDENGNYFQLREDWW DANRETVWKAITCNA
seq2 -----YEGLTTANGXKEYQDKNGGNFFKLREDWWTANRETVWKAITCGA
seq3 -----KRIYKKIFKEIHSGLSTKNGVKDRYQN-DGDNYFQLREDWWTANRSTVWKALTCSD
seq4 -----SQRHYKD-DGGNYFQLREDWWTANRHTVWEAITCSA
seq5 -----NVAALKTRYEK-DGQNFYFQLREDWWTANRATIWEAITCSA
seq6 -----FSKNIX-QIEELQDEWLLEARYKD-TDNYYELREHWWTENRHTVWEALTCEA
seq7 -----KELWEALCSR

seq1 --GGGKYFRNTCDG--GQNPTETQNNCRCIG-----ATVP TYFDYVPQYLRWSDE
seq2 P-GDASYFHATCDSGDGRGGAQAPHKRCRDG-----ANVVPTYFDYVPQFLRWPEE
seq3 KLSNASYFRATC--SDGQSGAQANNYCRCNGDKPDDDKP-NTDPPTYFDYVPQYLRWSEE
seq4 DKGNA-YFRRTCNSADGKSQS QARNQCRC---KDENGKN-ADQVPTYFDYVPQYLRWSEE
seq5 DKGNA-YFRATCNSADGKSQS QARNQCRC---KDENGXN-ADQVPTYFDYVPQYLRWSEE
seq6 P-GNAQYFRNACS---EGKTATKGKRCR CISGDP-----PTYFDYVPQYLRWSEE
seq7 P-KGANYFVYKLD----RPKFSSDRCGHNYNGDP-----LTNL DYVPQYLRWSDE
```

Sunday, October 21, 12

19

bioperl-run can run clustalw and many other programs

- The Run package (bioperl-run) provides wrappers for executing some 60 common bioinformatics applications (bioperl-run in the repository system Git, see link below)
 - Bio::Tools::Run::Alignment::clustalw
- There are several pieces to bioperl these are all listed here
- http://www.bioperl.org/wiki/Using_Git
 - bioperl-live Core modules including parsers and main objects
 - bioperl-run Wrapper modules around key applications
 - bioperl-ext Ext package has C extensions including alignment routines and link to staden IO library for sequence trace reads.
 - bioperl-pedigree
 - bioperl-microarray
 - bioperl-gui
 - bioperl-db

Sunday, October 21, 12

20

Smart Essential coding practices

- use strict; use warnings;
- Put all the hard stuff in subroutines so you can write clean subroutine calls.
- If you want to re-use a subroutine several times, put it in a module and re-use the module eg Util.pm
- #comments (ESC-; makes a comment in EMACS)
 - comment what a subroutine expects and returns
 - comment anything new to you or unusual
- Use the correct amount of indentation for loops, logic, subroutines

Sunday, October 21, 12

21

Coding strategy

- coding time = thinking/design (10%) + code writing (30%) + testing and debugging (60%)
- Re-use and modify existing code as much as possible
- Write your code in small pieces and test each piece as you go.
- Get some simple code running first.
- Use more complicated tools/code only if you need to
- Think about the big picture:
 - total time = coding time + run time + analysis time + writing up results
 - will speeding up your code take longer than waiting for it to complete? Your time is valuable
- Check your input data and your output data
 - are there unexpected characters, line returns (\r or \n ?), whitespace at the end of lines, spaces instead of tabs. You can use
 - % od -c mydatafile | more
 - are there missing pieces, duplicated IDs?
- use a small piece of (real or fake) data to test your code
- Is the output **exactly** what you expect?

Sunday, October 21, 12

22

gene_pred_pipe.pl (by Scott Cain) part I

```
#!/usr/bin/perl -w

use strict;

use Bio::DB::GenBank;
use Bio::Tools::Run::RepeatMasker;
use Bio::Tools::Run::Genscan;
use Bio::Tools::GFF;

my $acc = $ARGV[0]; # read argument from command line

# main functions in simple subroutines
my $seq_obj = acc_to_seq_obj($acc);
my $masked_seq = repeat_mask($seq_obj);
my @predictions = run_genscan($masked_seq);
predictions_to_gff(@predictions);
warn "Done!\n";
exit(0);
#-----
```

Sunday, October 21, 12

23

gene_pred_pipe.pl (by Scott Cain) part II

```
sub acc_to_seq_obj {
    #takes a genbank accession, fetches the seq from
    #genbank and returns a Bio::Seq object
    #parent script has to `use Bio::DB::Genbank` 
    my $acc = shift;
    my $db = new Bio::DB::GenBank;
    return $db->get_Seq_by_id($acc);
}

sub repeat_mask {
    #takes a Bio::Seq object and runs RepeatMasker locally.
    #Parent script must `use Bio::Tools::Run::RepeatMasker` 
    my $seq = shift;
    #BTRRM->new() takes a hash for configuration parameters
    #You'll have to set those up appropriately
    my $factory = Bio::Tools::Run::RepeatMasker->new();
    return $factory->masked_seq($seq);
}
```

Sunday, October 21, 12

24

gene_pred_pipe.pl (by Scott Cain) part III

```
sub run_genscan {
    #takes a Bio::Seq object and runs Genscan locally and returns
    #a list of Bio::SeqFeatureI objects
    #Parent script must `use Bio::Tools::Run::Genscan`
    my $seq = shift;
    #BTRG->new() takes a hash for configuration parameters
    #You'll have to set those up appropriately
    my $factory = Bio::Tools::Run::Genscan->new();
    #produces a list of Bio::Tools::Prediction::Gene objects
    #which inherit from Bio::SeqFeature::Gene::Transcript
    #which is a Bio::SeqFeatureI with child features
    my @genes = $factory->run($seq);
    my @features;
    for my $gene (@genes) {
        push @features, $gene->features;
    }
    return @features;
}
sub predictions_to_gff {
    #takes a list of features and writes GFF2 to a file
    #parent script must `use Bio::Tools::GFF`
    my @features = @_;
    my $gff_out = Bio::Tools::GFF->new(-gff_version => 2,
                                         -file           => '>prediction.gff');
    $gff_out->write_feature($_) for (@features);
    return;
}
```

Sunday, October 21, 12

25

Getting arguments from the command line with Getopt::Long and GetOptions()

- order of arguments doesn't matter
- deals with flags, integers, decimals, strings, lists
- complicated.pl -flag -c 4 --price 34.55 --name 'expensive flowers'

```
use Getopt::Long;
my ($flag, $count, $price, $string);
GetOptions( "flag" => \$flag,
            "c|count=i", \$count, # i means integer can use -c
                           # or --count on command line
            "price=f", \$price, # f means floating point
                           # number 0.12,3e-49
            "name=s", \$string, # s means string
                           # NOTE: always use trailing ','
# after last element so you can add more elements later
            );
# now $flag=1, $count=4, $price = 34.55,
# and $name = 'expensive flowers'
```

Sunday, October 21, 12

26

genbank_to_blast.pl (by Scott Cain) part I

```
#!/usr/bin/perl -w
use strict;
use lib "/home/scott/cvs_stuff/bioperl-live";    # this will change depending
                                                # on your machine
use Getopt::Long;
use Bio::DB::GenBank;
#use Bio::Tools::Run::RepeatMasker;    # running repeat masked first is a good
                                                # idea, but takes a while
use Bio::Tools::Run::RemoteBlast;
use Bio::SearchIO;
use Bio::SearchIO::Writer::GbrowseGFF;
use Bio::SearchIO::Writer::HTMLResultWriter;
use Data::Dumper;      # print out contents of objects etc
#take care of getting arguments
my $usage = "$0 [--html] [--gff] --accession <GB accession number>";
my ($HTML,$GFF,$ACC);
GetOptions ("html"        => \$HTML,
            "gff"         => \$GFF,
            "accession=s" => \$ACC);
unless ($ACC) {
    warn "$usage\n";
    exit(1);
}
#This will set GFF as the default if nothing is set but allowing both to be set
$GFF ||=1 unless $HTML;
#Now do real stuff ...
```

Sunday, October 21, 12

27

genbank_to_blast.pl (by Scott Cain) part II

```
# Now do real stuff
# nice and neat subroutine calls
# easy to understand logic of code
my $seq_obj      = acc_to_seq_obj($ACC);
my $masked_seq = repeat_mask($seq_obj);
my $blast_res   = blast_seq($masked_seq);
gff_out($blast_res, $ACC) if $GFF;
html_out($blast_res, $ACC) if $HTML;
#-----
```

Sunday, October 21, 12

28

genbank_to_blast.pl (by Scott Cain) part III

```
sub acc_to_seq_obj {
    print STDERR "Getting record from GenBank\n";
    my $acc = shift;
    my $db  = new Bio::DB::GenBank;
    return $db->get_Seq_by_id($acc);
}
sub repeat_mask {
    my $seq      = shift;
    return $seq;  #short circuiting RM since we
                  #don't have it installed, but this would be where
                  # you would run it
#    my $factory = Bio::Tools::Run::RepeatMasker-
>new();
#    return $factory->masked_seq($seq);
}
```

Sunday, October 21, 12

29

genbank_to_blast.pl (by Scott Cain) part IV

```
sub blast_seq {
    my $seq      = shift;
    my $prog    = 'blastn';
    my $e_val   = '1e-10';
    my $db      = 'refseq_rna';
    my @params = (
        -prog    => $prog,
        -expect  => $e_val,
        -readmethod => 'SearchIO',
        -data     => $db
    );
    my $factory = Bio::Tools::Run::RemoteBlast->new(@params);
    $factory->submit_blast($seq);
    my $v = 1; # message flag
    print STDERR "waiting for BLAST..." if ( $v > 0 );
    while ( my @rids = $factory->each_rid ) {
        foreach my $rid ( @rids ) {
            my $rc = $factory->retrieve_blast($rid);
            if( !ref($rc) ) { #waiting...
                if( $rc < 0 ) {
                    $factory->remove_rid($rid);
                }
                print STDERR "." if ( $v > 0 );
                sleep 25;
            }
            else {
                print STDERR "\n";
                return $rc->next_result();
            }
        }
    }
}
```

Sunday, October 21, 12

30

genbank_to_blast.pl (by Scott Cain) part V

```
sub gff_out {
    my ($result, $acc) = @_;
    my $gff_out = Bio::SearchIO->new(
        -output_format => 'GbrowseGFF',
        -output_signif => 1,
        -file           => ">$acc.gff",
        -reference     => 'query',
        -hsp_tag       => 'match_part',
    );
    $gff_out->write_result($result);
}
sub html_out {
    my ($result, $acc) = @_;
    my $writer = Bio::SearchIO::Writer::HTMLResultWriter->new();
    my $html_out = Bio::SearchIO->new(
        -writer => $writer,
        -format => 'blast',
        -file   => ">$acc.html"
    );
    $html_out->write_result($result);
}
```

Sunday, October 21, 12

31

HTML version of blast report: NM_000492.html Bioperl Reformatted HTML of BLASTN Search Report for NM_000492

BLASTN 2.2.12 [Aug-07-2005]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997), "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", Nucleic Acids Res. 25:3389-3402.

Query= NM_000492 Homo sapiens cystic fibrosis transmembrane conductance regulator, ATP-binding cassette (sub-family C, member 7) (CFTR), mRNA.

(6,129 letters)

Database: NCBI Transcript Reference Sequences

311,041 sequences; 606,661,208 total letters

Sequences producing significant alignments:	Score (bits)	E value
refNM_000492.2 Homo sapiens cystic fibrosis transmembrane conductance re...	1.201e+04	0
refNM_001032938.1 Macaca mulatta cystic fibrosis transmembrane conductance ...	8187	0
refNM_001007143.1 Canis familiaris cystic fibrosis transmembrane conductanc...	5019	0
refNM_174018.2 Bos taurus cystic fibrosis transmembrane conductance regu...	3253	0
refNM_001009781.1 Ovis aries cystic fibrosis transmembrane conductance regu...	3229	0
refNM_021050.1 Mus musculus cystic fibrosis transmembrane conductance re...	888	0
refXM_342645.2 PREDICTED: Rattus norvegicus cystic fibrosis transmembran...	714	0
refXM_347229.2 PREDICTED: Rattus norvegicus similar to cystic fibrosis t...	682	0

Sunday, October 21, 12

32

GFF output: NM_000492.gff

```
ref|NM_000492.2| BLASTN match 1 6129 1.201e+04 + . ID=match_sequence1;Target=EST:NM_000492+1+6129  
ref|NM_000492.2| BLASTN HSP 1 6060 + . ID=match_hsp1;Parent=match_sequence1;Target=EST:NM_000492+1+6129  
ref|NM_001032938.1| BLASTN match 1 4446 8187 + . ID=match_sequence2;Target=EST:NM_000492+133+4575  
ref|NM_001032938.1| BLASTN HSP 1 4446 4130 + . ID=match_hsp2;Parent=match_sequence2;Target=EST:NM_000492+133+4575  
ref|NM_001007143.1| BLASTN match 1 4332 5019 + . ID=match_sequence3;Target=EST:NM_000492+133+4455  
ref|NM_001007143.1| BLASTN HSP 1 4332 2532 + . ID=match_hsp3;Parent=match_sequence3;Target=EST:NM_000492+133+4455
```



```
ref|NM_000492.2| BLASTN match 1 6129 1.201e+04 + . ID=match_sequence1;Target=EST:NM_000492+1+6129  
ref|NM_000492.2| BLASTN HSP 1 6060 + . ID=match_hsp1;Parent=match_sequence1;Target=EST:NM_000492+1+6129  
ref|NM_001032938.1| BLASTN match 1 4446 8187 + . ID=match_sequence2;Target=EST:NM_000492+133+4575  
ref|NM_001032938.1| BLASTN HSP 1 4446 4130 + . ID=match_hsp2;Parent=match_sequence2;Target=EST:NM_000492+133+4575  
ref|NM_001007143.1| BLASTN match 1 4332 5019 + . ID=match_sequence3;Target=EST:NM_000492+133+4455  
ref|NM_001007143.1| BLASTN HSP 1 4332 2532 + . ID=match_hsp3;Parent=match_sequence3;Target=EST:NM_000492+133+4455  
ref|NM_174018.2| BLASTN match 54 5760 3253 + . ID=match_sequence4;Target=EST:NM_000492+133+4455  
ref|NM_174018.2| BLASTN HSP 54 2705 1641 + . ID=match_hsp4;Parent=match_sequence4;Target=EST:NM_000492+133+4455
```

Sunday, October 21, 12

33

How to approach perl pipelines

- use strict and warnings
- use (bio)perl as glue
- http://www.bioperl.org/wiki/Main_Page
- google.com
- test small pieces as you write them (debugger: perl -d)
- construct a command line and test it (catch failure ... or die...)
- convert into system call, check it worked with small sample dataset
- extend to more complex code only as needed
- if you use code more than once, put it into a subroutine in a module e.g. Util.pm
- get command line arguments with GetOptions()

Sunday, October 21, 12

34

Bioperl

Sofia Robb

What is Bioperl?

Collection of tools to help you get your work done

Open source, contributed by users

Used by GMOD, wormbase, flybase, me, you

<http://www.bioperl.org>

Why use BioPerl?

Code is already written.
Manipulate sequences.
Run programs (e.g., blast, clustalw and phylip).
Parsing program output (e.g., blast and alignments).
And much, much more. (<http://www.bioperl.org/wiki/Bptutorial.pl>)

3

Learning about bioperl

Manipulation of sequences from a file

Query a local fasta file

Creating a sequence record

File format conversions

Retrieving annotations

Parsing Blast output

Manipulating Multiple Alignments

Other Cool Things

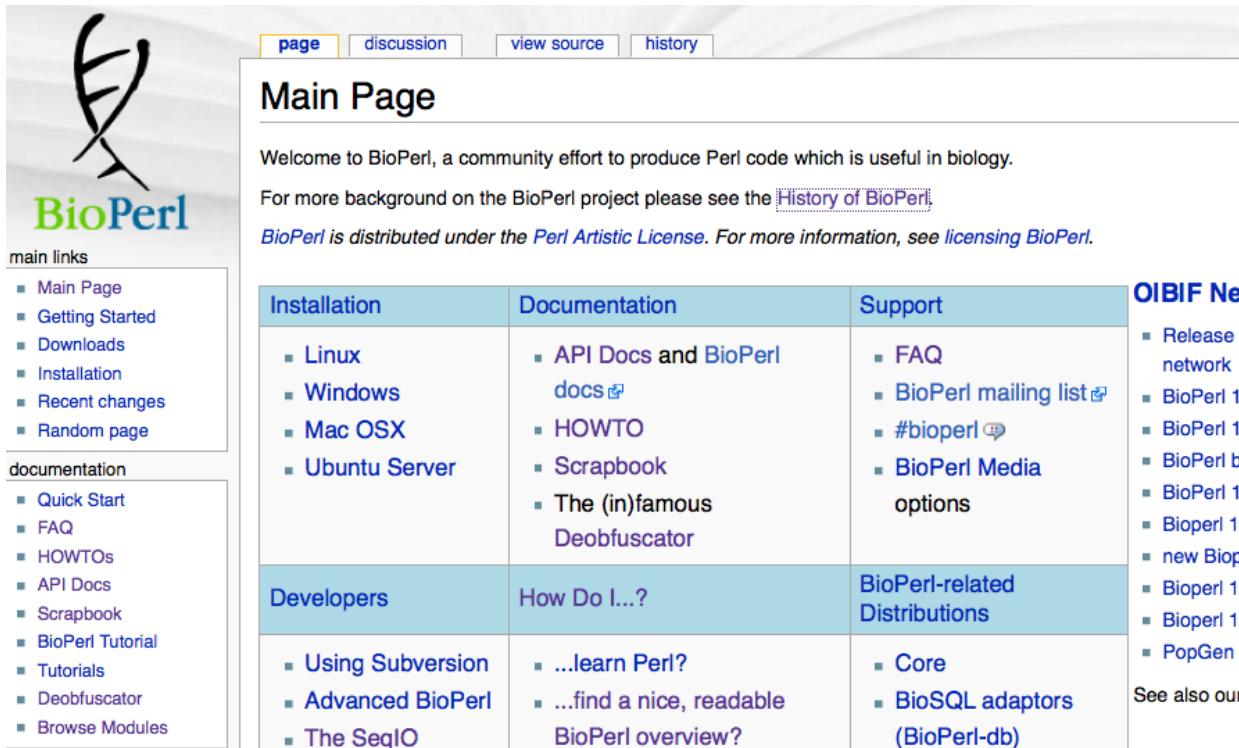
4

Learning about Bioperl:

Navigating Bioperl website
Deobfuscator
Bioperl docs

5

www.bioperl.org Main Page



The screenshot shows the main page of the BioPerl website. At the top right, there are four buttons: 'page' (highlighted in yellow), 'discussion', 'view source', and 'history'. The main content area has a large logo on the left and a title 'Main Page' at the top. Below the title, there is a welcome message and links to the 'History of BioPerl' and 'Perl Artistic License'. A sidebar on the left contains 'main links' and 'documentation' sections. The main content area features a table with three columns: 'Installation', 'Documentation', and 'Support'. The 'Documentation' column includes links to 'API Docs and BioPerl docs', 'HOWTO', 'Scrapbook', and 'The (in)famous Deobfuscator'. The 'Support' column includes links to 'FAQ', 'BioPerl mailing list', '#bioperl', and 'BioPerl Media options'. The 'Developers' section has a 'How Do I...?' link and a 'BioPerl-related Distributions' link. The 'OIBIF Net' sidebar on the right lists various BioPerl-related projects and links.

Installation	Documentation	Support
■ Linux ■ Windows ■ Mac OSX ■ Ubuntu Server	■ API Docs and BioPerl docs ■ HOWTO ■ Scrapbook ■ The (in)famous Deobfuscator	■ FAQ ■ BioPerl mailing list ■ #bioperl ■ BioPerl Media options
Developers	How Do I...?	BioPerl-related Distributions
■ Using Subversion ■ Advanced BioPerl ■ The SeqIO	■ ...learn Perl? ■ ...find a nice, readable BioPerl overview?	■ Core ■ BioSQL adaptors (BioPerl-db)

OIBIF Net

- Release 1 network
- BioPerl 1.
- BioPerl 1.
- BioPerl bi
- BioPerl 1.
- BioPerl 1.
- new Biop
- Bioperl 1.
- Bioperl 1.
- PopGen 1

See also our

6

 BioPerl

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

Search

Documentation

- Quick Start
- FAQ
- HOWTOs 
- API Docs
- Scrapbook
- Tutorials
- Deobfuscator
- Browse Modules

page discussion view source history

HOWTOs

HOWTOs are narrative-based descriptions of BioPerl modules focusing more on how to use them.

Contents [hide]

- 1 BioPerl HOWTOs
- 2 Requested HOWTOs
- 3 Copyright notice

BioPerl HOWTOs

Beginners HOWTO

An introduction to BioPerl, including reading and writing sequence files, running BLAST, and creating Perl objects.

SeqIO HOWTO

Sequence file I/O, with many script examples.

SearchIO HOWTO

Parsing reports from sequence comparison programs like BLAST and writing sequence files.

Tiling HOWTO

Using search reports parsed by SearchIO to obtain robust overall alignment.

BlastPlus HOWTO

Using BioPerl to create, manage, and query BLAST databases with the NCBI Blast+ interface.

7

 BioPerl

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs 
- API Docs
- Scrapbook
- BioPerl Tutorial
- Tutorials
- Deobfuscator
- Browse Modules

community

- News
- Mailing lists
- Supporting BioPerl

howto discussion view source history

HOWTO:Beginners

Contents [hide]

- 1 Authors
- 2 Copyright
- 3 Abstract
- 4 Introduction
- 5 Installing Bioperl
- 6 Getting Assistance
- 7 Perl Itself
- 8 Writing a script in Unix
- 9 Creating a sequence, and an Object
- 10 Writing a sequence to a file
- 11 Retrieving a sequence from a file
- 12 Retrieving a sequence from a database
- 13 Retrieving multiple sequences from a database
- 14 The Sequence Object
- 15 Example Sequence Objects
- 16 BLAST
- 17 Indexing for Fast Retrieval
- 18 More on Bioperl
- 19 Perl's Documentation System
- 20 The Basics of Perl Objects
- 21 A Simple Procedural Example
- 22 A Simple Object-Oriented Example

8



BioPerl

Main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

Documentation

- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial
- Tutorials
- Deobfuscator** ←
- Browse Modules

Community

- News
- Mailing lists
- Supporting BioPerl

Deobfuscator

Contents [hide]

- 1 What is the Deobfuscator?
- 2 Where can I find the Deobfuscator?
- 3 Have a suggestion?
- 4 Feature requests
- 5 Bugs

What is the Deobfuscator?

The Deobfuscator was written to make it easier to determine the methods that are available from a given BioPerl module (a common BioPerl FAQ).

BioPerl is a highly [object-oriented](#) software package, with often multiple levels of [inheritance](#). Although each individual module is usually well-documented for the methods specific to it, identifying the inherited methods is less straightforward.

The Deobfuscator indexes all of the [BioPerl](#) POD documentation, taking account of the inheritance tree (thanks to [Class::Inspector](#)), and then presents all of the methods available to each module through a searchable web interface.

Where can I find the Deobfuscator?

The Deobfuscator is currently available [here](#), indexing [bioperl-live](#).

9

Welcome to the BioPerl Deobfuscator

[[bioperl-live](#)]

[what is it?](#)

Search **class names** by string or Perl regex (examples: Bio::SeqIO, seq, fasta\$)

blast

OR select a class from the list:

Bio::SearchIO::blast	Event generator for event based parsing of blast reports
Bio::SearchIO::blast_pull	A parser for BLAST output
Bio::SearchIO::blasttable	Driver module for SearchIO for parsing NCBI -m 8/9 format
Bio::SearchIO::blastxml	A SearchIO implementation of NCBI Blast XML parsing.
Bio::SearchIO::megablast	a driver module for Bio::SearchIO to parse megablast reports (format 0)
Bio::Tools::Run::RemoteBlast	Object for remote execution of the NCBI Blast via HTTP
Bio::Tools::Run::StandAloneBlast	Object for the local execution of the NCBI BLAST program suite (blastall, blastpgp, bl2seq). There is experimental support for WU-Blast and NCBI rpsblast.
Bio::Tools::Run::StandAloneNCBIBlast	Object for the local execution of the NCBI BLAST program suite (blastall, blastpgp, bl2seq). With experimental support for NCBI rpsblast.

10

Deobfuscator

Bio::SearchIO::XML::BlastHandler	XML Handler for NCBI Blast XML parsing.
Bio::SearchIO::XML::PsiBlastHandler	XML Handler for NCBI Blast PSIBLAST XML parsing.

sort by method ▲

methods for Bio::Tools::Run::StandAloneBlast			
executable	Bio::Tools::Run::StandAloneBlast	string representing the full path to the exe	my \$exe = \$blastfactory->executable('blastn');
finally	Bio::Root::Root	not documented	not documented
io	Bio::Tools::Run::WrapperBase	Bio::Root::IO object	\$obj->io(\$newval)
new	Bio::Tools::Run::StandAloneBlast	Bio::Tools::Run::StandAloneNCBIBlast or StandAloneWUblast	my \$obj = Bio::Tools::Run::StandAloneBlast->new();
no_param_checks	Bio::Tools::Run::WrapperBase	value of no_param_checks	\$obj->no_param_checks(\$newval)
otherwise	Bio::Root::Root	not documented	not documented
outfile_name	Bio::Tools::Run::WrapperBase	string	my \$outfile = \$wrapper->outfile_name();
program	Bio::Tools::Run::StandAloneBlast	not documented	not documented

11

doc.bioperl.org

bio

Log in / create account

page discussion view source history

API Docs

Contents [hide]

- 1 Online POD Documentation
- 2 Documentation from the Deobfuscator
- 3 Documentation from the CPAN
- 4 Browsing Subversion repositories

Online POD Documentation

POD Documentation is available for bioperl-live and past releases at doc.bioperl.org.

Alternatively you can enter the module name in the search box and see any contributed Wiki documentation for the module.

Documentation from the Deobfuscator

The Deobfuscator indexes all of the BioPerl POD documentation, taking account of the inheritance tree, and then presents all of the methods available to each module through a [searchable web interface](#).

Documentation from the CPAN

←

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial
- Tutorials
- Deobfuscator

12



Perldoc (Pdoc rendered) documentation for BioPerl Modules

BioPerl

Released Code

Official documentation for released code is available here:

- BioPerl 1.6.0, download the entire doc set [here](#).
- BioPerl 1.5.2, download the entire doc set [here](#).
- BioPerl 1.5.1, download the entire doc set [here](#).
- BioPerl 1.4, download the entire doc set [here](#).
- BioPerl 1.2.3, download the entire doc set [here](#).
- BioPerl 1.2.2, download the entire doc set [here](#).
- BioPerl 1.2, download the entire doc set [here](#).
- BioPerl 1.0.2, download the entire doc set [here](#).
- BioPerl 1.0.1, download the entire doc set [here](#).
- BioPerl 1.0, download the entire doc set [here](#).

Active Code

This documentation represents the active development code and is autogenerated daily from the SVN repository:

Module	Description
■ bioperl-live	BioPerl Core Code
■ bioperl-corba-server	BioPerl BioCORBA Server Toolkit (wraps bioperl objects as BioCORBA objects and runs them in an ORBit ORB)
■ bioperl-corba-client	BioPerl BioCORBA Client Toolkit (wraps BioCORBA objects as bioperl objects)



All Modules TOC All

bioperl-live
bioperl-live::Bio
bioperl-live::Bio::Align
bioperl-live::Bio::AlignIO
bioperl-

PhyloNetwork

PrimarySeq

PrimarySeqI

PullParserI

Range

RangeI

SearchDist

SearchIO

Seq

SeqAnalysisParserI

SeqFeatureI

SeqI

SeqIO

SeqUtils

SimpleAlign

SimpleAnalysisI

Bio SeqIO

Summary	Included libraries	Package variables	Synopsis	Description	General documentation	Methods
-------------------------	------------------------------------	-----------------------------------	--------------------------	-----------------------------	---------------------------------------	-------------------------

Toolbar

[WebCvs](#)

Summary

Bio::SeqIO - Handler for SeqIO Formats

Package variables

Privates (from "my" definitions)

```
%valid_alphabet_cache;  
$Sentry = 0
```

Included modules

[Bio::Factory::FTLocationFactory](#)

[Bio::Seq::SeqBuilder](#)

[Bio::Tools::GuessSeqFormat](#)

[Symbol](#)

Inherit

[Bio::Factory::SequenceStreamI](#) [Bio::Root::IO](#) [Bio::Root::Root](#)

Synopsis



Bio::SeqIO module synopsis

doc.bioperl.org

Synopsis

```
use Bio::SeqIO;

$in  = Bio::SeqIO->new(-file => "inputfilename" ,
                      -format => 'Fasta');
$out = Bio::SeqIO->new(-file => ">outputfilename" ,
                      -format => 'EMBL');

while ( my $seq = $in->next_seq() ) {
    $out->write_seq($seq);
}

# Now, to actually get at the sequence object, use the standard Bio::Seq
# methods (look at Bio::Seq if you don't know what they are)

use Bio::SeqIO;

$in  = Bio::SeqIO->new(-file => "inputfilename" ,
                      -format => 'genbank');

while ( my $seq = $in->next_seq() ) {
    print "Sequence ",$seq->id, " first 10 bases ",
          $seq->subseq(1,10), "\n";
}

# The SeqIO system does have a filehandle binding. Most people find this
```



Bio::SeqIO module description

doc.bioperl.org

Description

Bio::SeqIO is a handler module for the formats in the SeqIO set (eg, Bio::SeqIO::fasta). It is the officially sanctioned way of getting at the format objects, which most people should use.

The **Bio::SeqIO** system can be thought of like biological file handles. They are attached to filehandles with smart formatting rules (eg, genbank format, or EMBL format, or binary trace file format) and can either read or write sequence objects (Bio::Seq objects, or more correctly, Bio::SeqI implementing objects, of which Bio::Seq is one such object). If you want to know what to do with a Bio::Seq object, read **Bio::Seq**.

The idea is that you request a stream object for a particular format. All the stream objects have a notion of an internal file that is read from or written to. A particular SeqIO object instance is configured for either input or output. A specific example of a stream object is the Bio::SeqIO::fasta object.

Each stream object has functions

```
$stream->next_seq();
```

and

```
$stream->write_seq($seq);
```



Bio::SeqIO method list

doc.bioperl.org

Methods		
new	Description	Code
newfh	Description	Code
fh	Description	Code
_initialize	No description	Code
next_seq	Description	Code
write_seq	Description	Code
alphabet	Description	Code
_load_format_module	Description	Code
_concatenate_lines	Description	Code
_filehandle	Description	Code
_guess_format	Description	Code
DESTROY	No description	Code
TIEHANDLE	Description	Code
READLINE	No description	Code

17

Bio::SeqIO new method description

doc.bioperl.org

Methods description

new	code	next	Top
<pre>Title : new Usage : \$stream = Bio::SeqIO->new(-file => \$filename, -format => 'Format') Function: Returns a new sequence stream Returns : A Bio::SeqIO stream initialised with the appropriate format Args : Named parameters: -file => \$filename -fh => filehandle to attach to -format => format Additional arguments may be used to set factories and builders involved in the sequence object creation. None of these must be provided, they all have reasonable defaults. -seqfactory the Bio::Factory::SequenceFactoryI object -locfactory the Bio::Factory::LocationFactoryI object -objbuilder the Bio::Factory::ObjectBuilderI object</pre>			

See [Bio::SeqIO::Handler](#)

18

Manipulation of sequences from a file

1c

Problem:

You have a sequence file and you want to do something to each sequence.

What do you do first?

HowTo:

<http://www.bioperl.org/wiki/HOWTOs>

2

[page](#) [discussion](#) [view source](#) [history](#)



BioPerl

main links

- [Main Page](#)
- [Getting Started](#)
- [Downloads](#)
- [Installation](#)
- [Recent changes](#)
- [Random page](#)

documentation

- [Quick Start](#)
- [FAQ](#)
- **[HOWTOs](#)** ←
- [API Docs](#)
- [Scrapbook](#)
- [BioPerl Tutorial](#)
- [Tutorials](#)
- [Deobfuscator](#)
- [Browse Modules](#)

OBDA Access HOWTO

HOWTOs

HOWTOs are narrative-based descriptions of BioPerl modules focusing more on a concept or a task than one specific module.

BioPerl HOWTOs

Beginners HOWTO ←

An introduction to BioPerl, including reading and writing sequence files, running and parsing BLAST, retrieving from databases, and more.

SeqIO HOWTO

Sequence file I/O, with many script examples.

SearchIO HOWTO

Parsing reports from sequence comparison programs like BLAST and writing custom reports.

Tiling HOWTO

Using search reports parsed by SearchIO to obtain robust overall alignment statistics

Feature-Annotation HOWTO

Reading and writing detailed data associated with sequences.

SimpleWebAnalysis HOWTO

Submitting sequence data to Web forms and retrieving results.

Flat Databases HOWTO

Indexing local sequence files for fast retrieval.

PAML HOWTO

Using the PAML package using BioPerl.

OBDA Access HOWTO

21

[howto](#) [discussion](#) [view source](#) [history](#)



BioPerl

main links

- [Main Page](#)
- [Getting Started](#)
- [Downloads](#)
- [Installation](#)
- [Recent changes](#)
- [Random page](#)

documentation

- [Quick Start](#)
- [FAQ](#)
- **[HOWTOs](#)** ←
- [API Docs](#)
- [Scrapbook](#)
- [BioPerl Tutorial](#)
- [Tutorials](#)
- [Deobfuscator](#)
- [Browse Modules](#)

community

- [News](#)
- [Mailing lists](#)
- [Supporting BioPerl](#)

HOWTO:Beginners

Contents [hide]

- 1 Authors
- 2 Copyright
- 3 Abstract
- 4 Introduction
- 5 Installing Bioperl
- 6 Getting Assistance
- 7 Perl Itself
- 8 Writing a script in Unix
- 9 Creating a sequence, and an Object
- 10 Writing a sequence to a file
- 11 Retrieving a sequence from a file ←
- 12 Retrieving a sequence from a database
- 13 Retrieving multiple sequences from a database
- 14 The Sequence Object
- 15 Example Sequence Objects
- 16 BLAST
- 17 Indexing for Fast Retrieval
- 18 More on Bioperl
- 19 Perl's Documentation System
- 20 The Basics of Perl Objects
- 21 A Simple Procedural Example
- 22 A Simple Object-Oriented Example

22

Retrieving a sequence from a file

One beginner's mistake is to not use `Bio::SeqIO` when working with sequence files. This is understandable in some respects. You may have read about Perl's `open` function, and Bioperl's way of retrieving sequences may look odd and overly complicated, at first. But don't use `open!` Using `open` immediately forces you to do the parsing of the sequence file and this can get complicated very quickly. Trust the SeqIO object, it's built to open and parse all the common [sequence formats](#), it can read and write to files, and it's built to operate with all the other Bioperl modules that you will want to use.

Let's read the file we created previously, "sequence.fasta", using SeqIO. The syntax will look familiar:

```
#!/bin/perl -w
use Bio::SeqIO;
$seqio_obj = Bio::SeqIO->new(-file => "sequence.fasta", -format => "fasta");
```

One difference is immediately apparent: there is no `>` character. Just as with the `open()` function this means we'll be reading from the "sequence.fasta" file. Let's add the key line, where we actually retrieve the Sequence object from the file using the `next_seq` method:

```
#!/bin/perl -w
use Bio::SeqIO;
$seqio_obj = Bio::SeqIO->new(-file => "sequence.fasta", -format => "fasta");
$seq_obj = $seqio_obj->next_seq;
```

2

Log in / Create account

page discussion view source history

HOWTOs

HOWTOs are narrative-based descriptions of [BioPerl](#) modules focusing more on a concept or a task than one specific module.

BioPerl HOWTOs

Beginners HOWTO

An introduction to [BioPerl](#), including reading and writing sequence files, running and parsing [BLAST](#), retrieving from databases, and more.

SeqIO HOWTO

Sequence file I/O, with many script examples.

SearchIO HOWTO

Parsing reports from sequence comparison programs like [BLAST](#) and writing custom reports.

Tiling HOWTO

Using search reports parsed by SearchIO to obtain robust overall alignment statistics

Feature-Annotation HOWTO

Reading and writing detailed data associated with sequences.

SimpleWebAnalysis HOWTO

Submitting sequence data to Web forms and retrieving results.

Flat Databases HOWTO

Indexing local sequence files for fast retrieval.

PAML HOWTO

Using the [PAML](#) package using [BioPerl](#).

OBDA Access HOWTO

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial

 BioPerl

2



- | Main Page
- | Getting Started
- | Downloads
- | Installation
- | Recent changes
- | Random page

- | Quick Start
- | FAQ
- | HOWTOs
- | API Docs
- | Scrapbook
- | BioPerl Tutorial
- | Tutorials
- | Deobfuscator
- | Browse Modules

- | News
- | Mailing lists
- | Supporting BioPerl
- | BioPerl Media
- | Hot Topics

HOWTO:SeqIO

This HOWTO will teach you about the Bio::SeqIO system for reading and writing sequences of various formats.

Contents [hide]

- 1 The basics
- 2 10 second overview
- 3 Background Information
- 4 Formats
- 5 Working Examples
- 6 To and From a String
- 7 And more examples...
- 8 Caveats
- 9 Error Handling
- 10 Speed, Bio::Seq::SeqBuilder

The basics

This section assumes you've never seen BioPerl before, perhaps you're a biologist trying to get some informal something about this hot topic, "bioinformatics". Your first script may want to get some information from a file c

A piece of advice: always use the module Bio::SeqIO! Here's what the first lines of your script might look like:

```
#!/bin/perl

use strict;
use Bio::SeqIO;

my $file = shift; # get the file name, somehow
my $seqio_object = Bio::SeqIO->new(-file => $file);
my $seq_object = $seqio_object->next_seq;
```

```
#!/usr/bin/perl -w
#file: inFasta_loop.pl
use strict;
use Bio::SeqIO;

my $file = shift;

my $seqIO_object = Bio::SeqIO->new(
    -file => $file,
    -format => 'fasta',
);

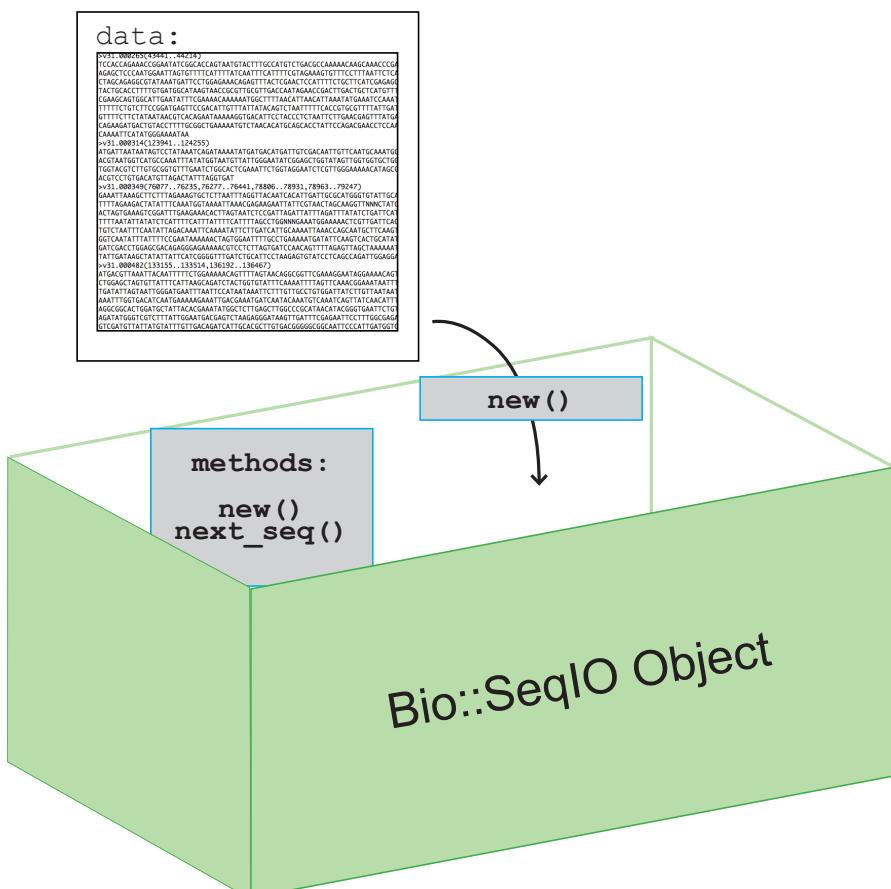
while (my $seq_object = $seqIO_object->next_seq) {
    #do stuff to each sequence in the fasta
}
```

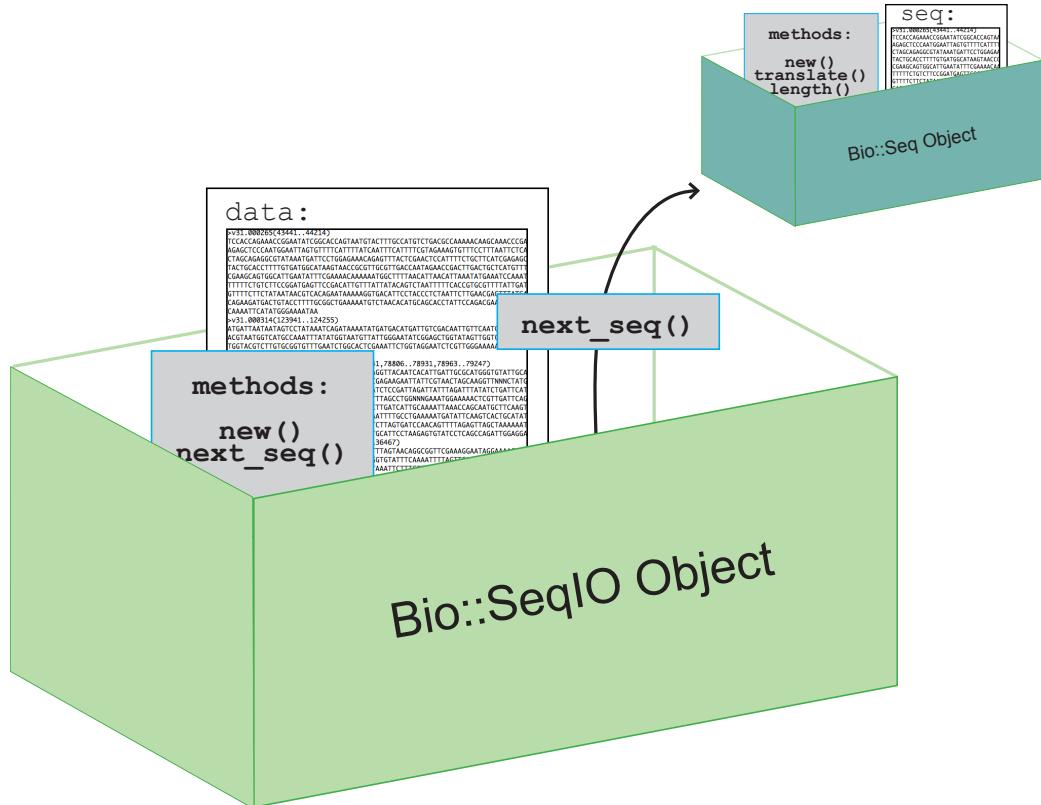
What is a SeqIO object?
What is a Seq object?

Objects

Objects are like boxes that hold
your data and
tools (methods) for your data

2





2

```

#!/usr/bin/perl -w
#file: inFasta_loop.pl
use strict;
use Bio::SeqIO;

# get fasta filename from user input
my $file = shift;

# create a SeqIO obj with $file as filename
# $seqIO_object contains all the individual sequence
#      that are in the file named $file
my $seqIO_object = Bio::SeqIO->new(
    -file => $file,
    -format => 'fasta',
);

# using while loop and next_seq method to "get to"
# and create a Seq obj for each individual sequence
# in the SeqIO obj of many sequences
while (my $seq_object = $seqIO_object->next_seq) {
    #do stuff to each sequence in the fasta
}

```

3

```

1. Get a file name from user
input (@ARGV) and stores in
$file
2. Create a new seqIO object
in $seqIO_object, using
filename $file and format
'fasta'
3. Create a second seqIO
object in $out using format
'fasta'
4. Loop thru each seq object
in $seqIO_object storing
information from the object in
variables.
5. Print out the stored
information
6. Print out $seq_object using
the method or tool 'write_
seq()' and the seqIO object
$out.

```

```

#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;

my $file = shift;
my $seqIO_object = Bio::SeqIO->new(
    -file => $file,
    -format => 'fasta',
);

my $out_seqIO_Obj = Bio::SeqIO->new(-format => 'fasta');

while (my $seq_object = $seqIO_object->next_seq){
    my $id = $seq_object->id;
    my $desc = $seq_object->desc;
    my $seqString = $seq_object->seq;
    my $revComp = $seq_object->revcom;
    my $Alphabet = $seq_object->alphabet;
    my $translation_seq_obj = $seq_object->translate;
    my $translation = $translation_seq_obj->seq;
    my $seqLen = $seq_object->length;

    print "translation: $translation\n";
    print "alphabet: $Alphabet\n";
    print "seqLen: $seqLen\n";

    #prints to STDOUT
    $out_seqIO_Obj->write_seq($seq_object);
}

```

31

fasta input:

<pre>>seqName seq description is blah blah blah AGGCTCAATTAGTTTCCTTGTCTTATTTAAAAGGTGTCCAGTG TGATGTGCAGCTGGTGGAGTCTGGGGGAGGCTTAGTGCAGCCTGGAG GGTCCCAGAAACTCTCCTGTGCAGCCTCTGGATTCACTTCAGTAGC TTTCCAATGCACGGGTTCTCAGGCTCCAGAGAAGGGGCTGGAGTG GGTCGCATACATTAGTAGTGGCAGTAGTACCCCTCCACTATGCAGACA CAGTGAAGGGCCGATTCAACCTCTCAAGAGACAATCCAAAGAACACC CTGTTCTGCAAATGACCAGTCTAAGGTCTGAGGAACACGCCATGTA TTACTGTGCAAGATGGGCTAACTACCCCTTAATGCTATGGACTACT GGGGTCAA</pre>	<pre>translation: RLNLVFLVLILKGVQCDVQLVESGGGLVQPGGSRKLSCAASGFTFSSF GMHWVRQAPEKGLEWVAYISSGSSTLHYADTVKGRFTISRDNPKNTLFLQMTSLRSEDTAN YYCARWGNYPYYAMDYWGQGTSVTVSS</pre>
---	--

Output:

```

alphapet: dna
seqLen: 408
>seqName seq description is blah blah blah
AGGCTCAATTAGTTTCCTTGTCTTATTTAAAAGGTGTCCAGTGATGTGCAGCTG
GTGGAGTCTGGGGGAGGCTTAGTGCAGCCTGGAGGGTCCCGGAAACTCTCCTGTGCAGCC
TCTGGATTCACTTCAGTAGCTTGAATGCACTGGGTTCTCAGGCTCCACTATGCAGACACAGTG
CTGGAGTGGGTCGCATACATTAGTAGTGGCAGTAGTACCCCTCCACTATGCAGACACAGTG
AAGGGCCGATTCAACCTCTCAAGAGACAATCCAAAGAACACCCTGTTCTGCAAATGACC
AGTCTAAGGTCTGAGGAACACGCCATGTATTACTGTGCAAGATGGGTAACTACCCTTAC
TATGCTATGGACTACTGGGGTCAAGGAACCTCAGTCACCGTCTCCTCA
```

32

Table from <http://www.bioperl.org/wiki/HOWTO:Beginners>

List of seq object methods

Table 1: Sequence Object Methods

Name	Returns	Example	Note
new	Sequence object	\$so = Bio::Seq->new(-seq => "MPQRAS")	create a new one, see Bio::Seq for more
seq	sequence string	\$seq = \$so->seq	get or set the sequence
display_id	identifier	\$so->display_id("NP_123456")	get or set an identifier
primary_id	identifier	\$so->primary_id(12345)	get or set an identifier
desc	description	\$so->desc("Example 1")	get or set a description
accession	identifier	\$acc = \$so->accession	get or set an identifier
length	length, a number	\$len = \$so->length	get the length
alphabet	alphabet	\$so->alphabet('dna')	get or set the alphabet ('dna','rna','protein')
subseq	sequence string	\$string = \$seq_obj->subseq(10,40)	Arguments are start and end
trunc	Sequence object	\$so2 = \$so1->trunc(10,40)	Arguments are start and end
revcom	Sequence object	\$so2 = \$so1->revcom	Reverse complement
translate	protein Sequence object	\$prot_obj = \$dna_obj->translate	See the Bioperl Tutorial for more
species	Species object	\$species_obj = \$so->species	See Bio::Species for more

3

Change 'format' in the new() method from 'fasta' to 'genbank' to change the way the SeqIO object \$out is displayed in STDOUT.

```
#title: inFasta_outGenBank.pl
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;

my $file = shift;
my $seqIO_object = Bio::SeqIO->new(
    -file => $file,
    -format => 'fasta',
);

my $out_seqIO_Obj= Bio::SeqIO->new(-format => 'genbank');

while (my $seq_object = $seqIO_object->next_seq){
    $out_seqIO_Obj->write_seq($seq_object); #prints to STDOUT
}

LOCUS      seqName          408 bp    dna     linear   UNK
DEFINITION seq description is blah blah blah
ACCESSION  unknown
FEATURES      Location/Qualifiers
BASE COUNT    95 a    98 c    111 g    104 t
ORIGIN
1 aggctcaatt tagtttcct tgtcattttt ttaaaagggtg tccagtgtga tggcgactg
61 gtggagtctg gggggggctt agtgcagcct ggagggtccc ggaaactctc ctgtcgaccc
121 tctggattca ctttcagtag ctttggaaatg cactgggttc gtcaggctcc agagaagggg
181 ctggagtggg tcgcatacat tagtagtggc agtagtaccc tccactatgc agacacatgt
241 aaggccat tcaccatctc aagagacaat cccaaagaaca ccctgttccct gcaaattacc
301 agtctaagggt ctgaggacac gggccatgtat tactgtcaa gatgggttaa ctacccttac
361 tatgctatgg actactgggg tcaaggaacc tcagtcaccg ttcctca
//
```

3

Query a local fasta file

3

Query a local fasta file

You have a fasta file that contains many records.

You want to retrieve a specific record.

You do not want to loop through all records until you find the correct record.

Use Bio::DB::Fasta.

3



BioPerl

Main links

- | Main Page
- | Getting Started
- | Downloads
- | Installation
- | Recent changes
- | Random page

Documentation

- | Quick Start
- | FAQ
- | HOWTOs
- | API Docs
- | Scrapbook
- | BioPerl Tutorial
- | Tutorials
- | Deobfuscator
- | Browse Modules

Community

- | News
- | Mailing lists
- | Supporting BioPerl
- | BioPerl Media

Deobfuscator

Contents [hide]

- 1 What is the Deobfuscator?
- 2 Where can I find the Deobfuscator?
- 3 Have a suggestion?
- 4 Feature requests
- 5 Bugs

What is the Deobfuscator?

The Deobfuscator was written to make it easier to determine the methods that are available from a given BioPerl module (a common BioPerl FAQ).

BioPerl is a highly object-oriented software package, with often multiple levels of inheritance. Although each individual module is usually well-documented for the methods specific to it, identifying the inherited methods is less straightforward.

The Deobfuscator indexes all of the BioPerl POD documentation, taking account of the inheritance tree (thanks to [Class::Inspector](#)), and then presents all of the methods available to each module through a searchable web interface.

Where can I find the Deobfuscator?

The Deobfuscator is currently available [here](#), indexing [bioperl-live](#).

3

Welcome to the BioPerl Deobfuscator

[bioperl-live]

Search **class names** by string or Perl regex (examples: Bio::SeqIO, seq, fasta\$)

fasta

3

OR select a class from the list:

Bio::AlignIO::fasta	fasta MSA Sequence input/output stream
Bio::AlignIO::largemultifasta	Largemultifasta MSA Sequence input/output stream
Bio::AlignIO::metfasta	Metfasta MSA Sequence input/output stream
Bio::DB::Fasta	Fast indexed access to a directory of fasta files
Bio::DB::Flat::BDB::fasta	fasta adaptor for Open-bio standard BDB-indexed flat file
Bio::Index::Fasta	Interface for indexing (multiple) fasta files
Bio::Search::HSP::FastaHSP	HSP object for FASTA specific data
Bio::Search::Hit::Fasta	Hit object specific for Fasta-generated hits
Bio::SearchIO::fasta	A SearchIO parser for FASTA results
Bio::Seq::SeqFastaSpeedFactory	Instantiates a new Bio::PrimarySeqI (or derived class) through a factory

sort by method

3

Bio::AlignIO::metfasta	Metfasta MSA Sequence input/output stream
Bio::DB::Fasta	Fast indexed access to a directory of fasta files
Bio::DB::Flat::BDB::fasta	fasta adaptor for Open-bio standard BDB-indexed flat file
Bio::Index::Fasta	Interface for indexing (multiple) fasta files
Bio::Search::HSP::FastaHSP	HSP object for FASTA specific data
Bio::Search::Hit::Fasta	Hit object specific for Fasta-generated hits
Bio::SearchIO::fasta	A SearchIO parser for FASTA results
Bio::Seq::SeqFastaSpeedFactory	Instantiates a new Bio::PrimarySeqI (or derived class) through a factory

sort by method

methods for **Bio::DB::Fasta**

Method	Class	Returns	Usage
alphabet	Bio::DB::Fasta	not documented	not documented
basename	Bio::DB::Fasta	not documented	not documented
calculate_offsets	Bio::DB::Fasta	not documented	not documented
caloffset	Bio::DB::Fasta	not documented	not documented
carp	Bio::Root::RootI	not documented	not documented
CLEAR	Bio::DB::Fasta	not documented	not documented
confess	Bio::Root::RootI	not documented	not documented
dbmargs	Bio::DB::Fasta	not documented	not documented
debut	Bio::Root::Root	none	\$obj->debut("This is debutting output");

BIO::DB::Fasta

Other packages in the module: [Bio::DB::Fasta](#) [Bio::PrimarySeq::Fasta](#)

[Summary](#) [Included libraries](#) [Package variables](#) [Synopsis](#) [Description](#)

Toolbar

[WebCvs](#)

Summary

Bio::DB::Fasta -- Fast indexed access to a directory of fasta files

Package variables

No package variables defined.

Included modules

[AnyDBM_File](#)

[Fcntl](#)

[File::Basename](#) [qw](#) (basename dirname)

[IO::File](#)

Inherit

[Bio::DB::SeqI](#) [Bio::Root::Root](#)

Synopsis

```
use Bio::DB::Fasta;

# create database from directory of fasta files
my $db      = Bio::DB::Fasta->new('/path/to/fasta/files');
```

Bio::DB::fasta module synopsis

doc.bioperl.org

Synopsis

```
use Bio::DB::Fasta;

# create database from directory of fasta files
my $db      = Bio::DB::Fasta->new('/path/to/fasta/files');

# simple access (for those without Bioperl)
my $seq      = $db->seq('CHROMOSOME_I',4_000_000 => 4_100_000);
my $revseq   = $db->seq('CHROMOSOME_I',4_100_000 => 4_000_000);
my @ids     = $db->ids;
my $length   = $db->length('CHROMOSOME_I');
my $alphabet = $db->alphabet('CHROMOSOME_I');
my $header   = $db->header('CHROMOSOME_I');

# Bioperl-style access
my $db      = Bio::DB::Fasta->new('/path/to/fasta/files');

my $obj      = $db->get_Seq_by_id('CHROMOSOME_I');
my $seq      = $obj->seq; # sequence string
my $subseq   = $obj->subseq(4_000_000 => 4_100_000); # string
my $strunc   = $obj->trunc(4_000_000 => 4_100_000); # seq object
my $length   = $obj->length;
# (etc)

# Bio::SeqIO-style access
my $stream  = Bio::DB::Fasta->new('/path/to/files')->get_PrimarySeq_stream;
while (my $seq = $stream->next_seq) {
    # Bio::PrimarySeqI stuff
}
```

41

Bio::DB::fasta module description

doc.bioperl.org

Description

Bio::DB::Fasta provides indexed access to one or more Fasta files. It provides random access to each sequence entry, and to subsequences within each entry, allowing you to retrieve portions of very large sequences without bringing the entire sequence into memory.

When you initialize the module, you point it at a single fasta file or a directory of multiple such files. The first time it is run, the module generates an index of the contents of the file or directory using the AnyDBM module (Berkeley DB* preferred, followed by GDBM_File, NDBM_File, and SDBM_File). Thereafter it uses the index file to find the file and offset for any requested sequence. If one of the source fasta files is updated, the module reindexes just that one file. (You can also force reindexing manually). For improved performance, the module keeps a cache of open filehandles, closing less-recently used ones when the cache is full.

The fasta files may contain any combination of nucleotide and protein sequences; during indexing the module guesses the molecular type. Entries may have any line length up to 65,536 characters, and different line lengths are allowed in the same file. However, within a sequence entry, all lines must be the same length except for the last.

4

Bio::DB::fasta method description

doc.bioperl.org

get_Seq_by_id	code	prev
Title : get_Seq_by_id Usage : my \$seq = \$db->get_Seq_by_id(\$id) Function: Bio::DB::RandomAccessI method implemented Returns : Bio::PrimarySeqI object Args : id		

4:

Query a local fasta file

```
#!/usr/bin/perl -w
use strict;
use Bio::DB::Fasta;

my $dbfile = 'uniprot_sprot.fasta';
my $db_obj = Bio::DB::Fasta->new($dbfile);

# retrieve a sequence
my $id = 'sp|Q13547|HDAC1_HUMAN';
my $seq_obj = $db_obj->get_Seq_by_id($id);

if ( $seq_obj ) {
    print "seq: ",$seq_obj->seq,"\n";
} else {
    warn("Cannot find $id\n");
}
```

Output

```
seq: MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIRMTHNLLNYGLYRKMEIYRPHKANAE
EMTKYHSDDYIKFLRSIRPDNMSEYSKQMQRFNVEDCPVFDGLFEFCQLSTGGSVASAVKLNKQQT
DIAVNWAGGLHHAKKSEASGFCYVNDIVLAILELLKYHQRVLYIDIDIHHGDGVEEAFYTTDRVMTV
SFHKYGEYFPGTGDLRDIGAGKGKYYAVNYPLRDGIDDESYEAIKPVMSKVMEMFQPSAVVLQCGS
DSLSGDRLGCFNLTIKGHAKCVEFKVSFNLPMLMLGGGGYTIRNVARCWTYETAVALTEIPNELPY
NDYFEYFGPDFKLHISPSNMTNQNTNEYLEKIKQRLFENLRMLPHAPGVQMQAIPEDAYPEESGDED
EDDPDKRISICSSDKRIACEEEFSDEEEEGEGGRKNSSNFKKAKRVKTEDEKEKDPEEKKEVTEEEK
TKEEVDEAVCVKEEVKIA
```

4

Creating a sequence record

4€

Creating a sequence record

You have a sequence and want to create a Seq object
on the fly.

Use Bio::Seq.

4€

Create a sequence record on the fly.

```
#!/usr/bin/perl -w  
use strict;  
use Bio::Seq;  
use Bio::SeqIO;  
  
#file:createSeqOnFly.pl  
  
my $seqObj = Bio::Seq->new(-seq => 'ATGAATGATGAA',  
                           -display_id => 'seq_example',  
                           -description=> 'this seq is awesome');  
  
my $out_seqIO_Obj = Bio::SeqIO->new(-format => 'fasta');  
$out_seqIO_Obj->write_seq($seqObj);  
  
print "Id: ", $seqObj->display_id, "\n";  
print "Length: ", $seqObj->length, "\n";  
print "Seq: ", $seqObj->seq, "\n";  
print "Subseq (3..6): ", $seqObj->subseq(3,6), "\n";  
print "Translation: ", $seqObj->translate->seq, "\n";
```

1. Create a new seq object

2. Create and print a new seqIO object in fasta format using \$seqObj

3. Get features of \$seqObj by using seqObj methods

Notice the coupling of methods.

4

Output

```
>seq_example this seq is awesome  
ATGAATGATGAA  
Id: seq_example  
Length: 12  
Seq: ATGAATGATGAA  
Subseq (3..6): GAAT  
Translation: MNDE
```

4

File format conversions

4

File format conversions

You have GenBank files and want to extract only the sequence in fasta format.

Use Bio::SeqIO.

5

Formats

BioPerl's SeqIO system understands lot of formats and can interconvert all of them. Here is a current listing of formats, as of version 1.5.

Table 1: Bio::SeqIO modules and formats supported

Name	Description	File extension	Module
abi	ABI tracefile	ab[i1]	Bio::SeqIO::abi
ace	Ace database	ace	Bio::SeqIO::ace
agave	AGAVE XML		Bio::SeqIO::agave
alf	ALF tracefile	alf	Bio::SeqIO::alf
asciitree	write-only, to visualize features		Bio::SeqIO::asciitree
bsml	BSML, using XML::DOM ↗	bsml	Bio::SeqIO::bsml
bsml_sax	BSML, using XML::SAX ↗		Bio::SeqIO::bsml_sax
chadoxml	CHADO sequence format		Bio::SeqIO::chadoxml
chaos	CHAOS sequence format		Bio::SeqIO::chaos
chaosxml	Chaos XML		Bio::SeqIO::chaosxml
cif	CIF tracefile	cif	Bio::SeqIO::cif

<http://www.bioperl.org/wiki/HOWTO:SeqIO>

51

```

OCUS      MUSIGHBAI          408 bp   mRNA   linear   ROD 27-APR-1993
DEFINITION Mouse Ig active H-chain V-region from MOPC21, subgroup VH-II,
mRNA.
ACCESSION J00522
VERSION J00522.1 GI:195052
KEYWORDS constant region; immunoglobulin heavy chain; processed gene; variable re-
ion; variable region subgroup VH-II.
SOURCE Mus musculus (house mouse).
ORGANISM Mus musculus
Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
Mammalia; Eutheria; Euarchontoglires; Glires; Rodentia;
Sciurognathi; Muroidea; Muridae; Murinae; Mus.
REFERENCE 1 (bases 1 to 408)
AUTHORS Bothwell,A.L., Paskind,M., Reth,M., Imanishi-Kari,T., Rajewsky,K.
and Baltimore,D.
TITLE Heavy chain variable region contribution to the NPb family of
antibodies: somatic mutation evident in a gamma 2a variable region
JOURNAL Cell 24 (3), 625-637 (1981)
PUBMED 6788376
COMMENT Original source text: Mouse C57Bl/6 myeloma MOPC21, cDNA to mRNA,
clone pAB-gamma-1-4. [1] studies the response in C57Bl/6 mice to
NP proteins. It is called the b-NP response because this mouse
strain carries the b-IgH haplotype. See other entries for b-NP
response for more comments.
FEATURES Location/Qualifiers
source 1..408
/db_xref="taxon:10090"
/mol_type="mRNA"
/organism="Mus musculus"
<1..408
</source>
<1..408
/CDS
<1..408
/db_xref="GI:195055"
/codon_start=1
/protein_id="AAD15290.1"
/translation="RLNLVFLVLILKGVQCDVQLVESGGGLVQPSSRKLSAACSGFT
FSSFGMHWRQAPEKGLEWVAYISSGSSTLHYADTVKGRFTISRDNPKNTLFLQMTSL
RSEDTAMYCARWNYPYAMDYWGQGTSTVSS"
/note="Ig H-chain V-region from MOPC21"
<1..48
<49..3408
<343..344
<356..357
</CDS>
<1..48
<49..3408
<product="Ig H-chain V-region from MOPC21 mature peptide"
<343..344
<note="V-region end/D-region start (+/- 1bp)"
<356..357
<note="D-region end/J-region start"
</FEATURES>
ASE COUNT      95 a     98 c     111 g    104 t
RIGIN      57 bp upstream of PvuII site, chromosome 12.
1 aggtcaatt tagtttctc tgtccattt ttaaaagggt tccagtgtga tggcagctg
61 gtggactctg gggggggatc agtgccgttcc ggaaaccttcc ctgtcgaccc
121 tctggatttc ctttcgttagt ctgtggatg cactgggtc gtccaggctcc agagaagggg
181 ctggatgtgg tcgcatacat tagtagtggc agtagtacccttccactatgc agacacagtg
241 aaggggccatc tccatccatc aagagacaa cccaaaggatc cccgtttttt gcaatgtacc
301 agtcttaagggt ctggggacac ccgttgcata gatgtttttt gatgtttttt ctacccttac
361 tatgttatgg actactgggg tcaaggaaacc tcgttcaccc ttcccttac

```

= GenBank Format



Fasta Format

```

>MUSIGHBAI Mouse Ig active H-chain V-region from MOPC21,
subgroup VH-II, mRNA.
AGGCCTAATTTAGTTTCCCTTATTAAAGGTGTCAGTGATGTGAGCTG
GTGGAGCTCTGGGGAGGCTTACTGCAGCTGAGGGTCCCGAAACTCTCTGTCAGCC
TCTGGAGTCACTTCACTAGTGGCAATGCACTGGCTCAGGCTCAGAGAAAGGGG
CTGGAGTGGTCCCATACATAGTAGTGGCAATGCACTATGCAGACACAGTG
AAGGGGGATTCACCATCTCAAGAGACAATCCCAAAGAACACCCCTGTTCTGCAAATGACC
AGTCTAAAGGCTGAGGGACAGGCCATGTATTACTGTGCAAGATGGGTAACTACCCCTTAC
TATGCTATGGACTACTGGGGTCAAGGAACCTCAGTCACCGTCTCCCTA

```

52

Convert from GenBank to fasta.

```
#!/usr/bin/perl -w  
use strict;  
use Bio::SeqIO;  
  
my ($informat,$outformat) = ('genbank','fasta');  
my ($infile,$outfile) = @ARGV;  
  
my $in_seqIO_Img = Bio::SeqIO->new(  
    -format => $informat,  
    -file => $infile,  
);  
my $out_seqIO_Img = Bio::SeqIO->new(  
    -format => $outformat,  
    -file => ">$outfile"  
);  
  
while ( my $seqObj = $in_seqIO_Img->next_seq ) {  
    $out_seqIO_Img->write_seq($seqObj);  
}  
#file:convert_genbank2fasta.pl
```

5

Retrieving annotations

5

Retrieving annotations

You have GenBank files and want to retrieve annotations.

Use Bio::SeqIO.

5

Sample GenBank file with Features/Annotations

LOCUS MUSIGHB1 408 bp mRNA linear ROD 27-APR-1993
DEFINITION Mouse Ig active H-chain V-region from MOPC21, subgroup VH-II, mRNA.
ACCESSION J00522
VERSION J00522.1 GI:195052
KEYWORDS constant region; immunoglobulin heavy chain; processed gene; variable region; variable region subgroup VH-II.
SOURCE Mus musculus (house mouse).
ORGANISM Mus musculus
Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
Mammalia; Eutheria; Euarchontoglires; Glires; Rodentia;
Sciurognathi; Muroidea; Muridae; Murinae; Mus.
REFERENCE 1 (bases 1 to 408)
AUTHORS Bothwell,A.L., Paskind,M., Reth,M., Imanishi-Kari,T., Rajewsky,K. and Baltimore,D.
TITLE Heavy chain variable region contribution to the NPb family of antibodies: somatic mutation evident in a gamma 2a variable region
JOURNAL Cell 24 (3), 625-637 (1981)
PUBMED 6788376
COMMENT Original source text: Mouse C57Bl/6 myeloma MOPC21, cDNA to mRNA, clone pAB-gamma-1-4. [1] studies the response in C57Bl/6 mice to NP proteins. It is called the b-NP response because this mouse strain carries the b-IgH haplotype. See other entries for b-NP response for more comments.

FEATURES Location/Qualifiers
source 1..408
/db_xref="taxon:10090"
/mol_type="mRNA"
/organism="Mus musculus"
CDS <1..>408
/db_xref="GI:195055"
/codon_start=1
/protein_id="AAD15290.1"
/translation="RLNLVLFLVILKGVQCDVQLVESGGGLVQPGGSRKLSCAAASGFT
FSSFGMHWRQAPEKGLEWVAYISSGSSLHYADTVKGRFTISRDNPFKNTLFLQMTSL
RSEDTAMYCCARWGNYYAMDYWQGTSTSFTVVS"
/note="Ig H-chain V-region from MOPC21"
sig_peptide <1..48
mat_peptide 49..>408
/product="Ig H-chain V-region from MOPC21 mature peptide"
misc_recomb 343..344
/note="V-region end/D-region start (+/- 1bp)"
misc_recomb 356..357
/note="D-region end/J-region start"
BASE COUNT 95 a 98 c 111 g 104 t
ORIGIN 57 bp upstream of Pvull site, chromosome 12.
1 aggtcaatt tagtttctt tgccttatt taaaagggtc tccatgttgta tggtcagctg
61 gtggatgtt gggggaggctt atggacggcc ggaaatgttcc ggaatcttcc ctgtgcggcc
121 ttctggatca ctttgcgttactt ctgttggatc acgtgggtt cttcggttcc agaaaggagg
181 ctggatgttg tggcatataact tagttatggt agtagtaccaccc tccatctgc agacacatgt
241 aaggccgttccatcaccc agaaagaca cccaaagaca cctgttccat gcaaataatggc
301 agtctaaatgtt ccgtggacac gggcaatgttacttgcgttccatgggttcaatcccttac
361 tcataatgtt acttccatgg tccaaatgttacttgcgttccatgggttcaatcccttac

5

```

FEATURES          Location/Qualifiers
source           1..408
                 /db_xref="taxon:10090"
                 /mol_type="mRNA"
                 /organism="Mus musculus"
CDS              <1..>408
                 /db_xref="GI:195055"
                 /codon_start=1
                 /protein_id="AAD15290.1"
                 /translation="RLNLVFLVLILKGVQCDVQLVESGGGLVQPSSRKLSCAASGFT
FSSFGMHWVRQAPEKGLEWVAYISSGSSTLHYADTVKGRTISRDNPKNTLFLQMTSL
RSEDTAMYYCARWGNYPYYAMDYWGQGTSVTVSS"
                 /note="Ig H-chain V-region from MOPC21"
sig_peptide      <1..48
mat_peptide      49..>408
                 /product="Ig H-chain V-region from MOPC21 mature peptide"
misc_recomb      343..344
                 /note="V-region end/D-region start (+/- 1bp)"
misc_recomb      356..357
                 /note="D-region end/J-region start"

```

5

```

#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;

my $infile = shift;
my $seqIO = Bio::SeqIO->new(
    -file => $infile,
    -format => 'genbank',
);
while (my $seqObj = $seqIO -> next_seq){
    my $name = $seqObj -> id;
    foreach my $feature_obj ($seqObj->get_SeqFeatures())

```

#file: get_annot_from_genbank.pl

```

        my $primary_tag = $feature_obj->primary_tag;
        my ($start, $end) = ($feature_obj->start, $feature_obj->end);
        my $range = $start . " .. " . $end;
        foreach my $tag ( sort $feature_obj->get_all_tags ) {
            my @values = $feature_obj->get_tag_values($tag);
            my $value_str = join ", ", @values;
            print "$name($range)\t$primary_tag\t$tag:$value_str\n";
        }
    }
}

```

get_SeqFeature
produces an array of
Bio::SeqFeature objects

Output

MUSIGHBA1 (1..408)	source	db_xref:taxon:10090
MUSIGHBA1 (1..408)	source	mol_type:mRNA
MUSIGHBA1 (1..408)	source	organism:Mus musculus
MUSIGHBA1 (1..408)	CDS	codon_start:1
MUSIGHBA1 (1..408)	CDS	db_xref:GI:195055
MUSIGHBA1 (1..408)	CDS	note:Ig H-chain V-region from MOPC21
MUSIGHBA1 (1..408)	CDS	protein_id:AAD15290.1
MUSIGHBA1 (1..408)	CDS	translation:RLNLVFLVLILKGVQCDVQLVESGGGLVQPSSRKLSCAASGFTFSSF FSSFGMHWVRQAPEKGLEWVAYISSGSSTLHYADTVKGRTISRDNPKNTLFLQMTSLRSEDTAMYYCARWGNYPYYAMDYWGQGTSVTVSS
MUSIGHBA1 (49..408)	mat_peptide	product:Ig H-chain V-region from MOPC21 mature peptide
MUSIGHBA1 (343..344)	misc_recomb	note:V-region end/D-region start (+/- 1bp)
		note:D-region end/J-region start

5

Manipulating Multiple Alignments

5

Use Bio::AlignIO

for parsing and writing multiple alignment file formats
including:

fasta, phylip, nexus, clustalw, msf, mega,
meme, pfam, psi, selex, stockholm.

6

Convert from fasta_aln to nexus

#file: multi_align_convert.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::AlignIO;

my $align_fasta = shift;
my $in_alignIO_obj = Bio::AlignIO->new(
    -format => 'fasta',
    -file => $align_fasta
);
my $out_alignIO_obj = Bio::AlignIO->new(
    -format => 'nexus',
    -file => ">$align_fasta.nex"
);
while( my $align_obj = $in_alignIO_obj->next_aln ){
    $out_alignIO_obj->write_aln($align_obj);
}
```

next_aln produces a Bio::SimpleAlign object



61

Bio::SimpleAlign Object

Remove some sequences and rewrite the result

Extract or remove columns

Calculate consensus string and percent identity

62

Parsing BLAST Output

6:

Parsing BLAST reports

Use Bio::SearchIO

6:

Where do you start?



BioPerl

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs ←
- API Docs
- Scrapbook
- BioPerl Tutorial
- Tutorials
- Deobfuscator
- Source Modules

Contents [hide]

- 1 Authors
- 2 Copyright
- 3 Abstract
- 4 Introduction
- 5 Installing Bioperl
- 6 Getting Assistance
- 7 Perl Itself
- 8 Writing a script in Unix
- 9 Creating a sequence, and an Object
- 10 Writing a sequence to a file
- 11 Retrieving a sequence from a file
- 12 Retrieving a sequence from a database
- 13 Retrieving multiple sequences from a database
- 14 The Sequence Object
- 15 Example Sequence Objects
- 16 BLAST ←



Here's an example of how one would use SearchIO to extract data from a BLAST report:

```
use Bio::SearchIO;
my $report_obj = new Bio::SearchIO(-format => 'blast',
                                  -file   => 'report.bls');
while( $result = $report_obj->next_result ) {
    while( $hit = $result->next_hit ) {
        while( $hsp = $hit->next_hsp ) {
            if ( $hsp->percent_identity > 75 ) {
                print "Hit\t", $hit->name, "\n", "Length\t", $hsp->length('total'),
                      "\n", "Percent_id\t", $hsp->percent_identity, "\n";
            }
        }
    }
}
```



BioPerl

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs

howto discussion view source history

HOWTO:SearchIO

Abstract

This is a HOWTO about the `Bio::SearchIO` system, how to use it, and how one goes about writing new adaptors to different output formats. We will also describe how the `Bio::SearchIO::Writer` modules work for outputting various formats from `Bio::Search` objects.

Contents [hide]

- 1 Abstract
 - 1.1 Authors
- 2 Background
- 3 Design
- 4 Parsing with Bio::SearchIO
 - 4.1 Avoiding possible confusion
 - 4.2 Using SearchIO



NCBI BLAST Report

Result

Hit

HSP

Result

6

BLASTX 2.2.12 [Aug-07-2005]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997), "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", Nucleic Acids Res. 25:3389-3402.

Query= smed-HDAC1-1
(1213 letters)

Database: swissprot_aa
427,028 sequences; 157,875,145 total letters

Searching.....done

Score	E
(bits)	Value
535	e-151

Sequences producing significant alignments:

sp|P56517|HDAC1_CHICK RecName: Full=Histone deacetylase 1; Short... 535 e-151

>sp|P56517|HDAC1_CHICK RecName: Full=Histone deacetylase 1; Short=HD1
length = 480

Score = 535 bits (1379), Expect = e-151
Identities = 255/343 (74%), Positives = 292/343 (85%), Gaps = 1/343 (0%)
Frame = +3

Query: 3 CPVFDGLPEFCQLSAGGSVASAVKLNKNAKADIAINWSGGLHHAKKSEASGF CYVNNDIVMG 182
CPVFDGLPEFCQLSAGGSVASAVKLNK + DIA+NW+GCLHHAKKSEASGF CYVNNDIV...
Sbjct: 100 CPVFDGLPEFCQLSAGGSVASAVKLNKQQTIDIAVNAGGLHHAKKSEASGF CYVNNDIVLA 159

Query: 183 ILELLKYHHERVLVYDIDIHHGDGVEEAFTTDRVMTVSFHKYGEYFPXXXXXX 362
Sbjct: 160 ILELLKYHQRVLYIIDIHHGDGVEEAFTTDRVMTVSFHKYGEYFPCTGDLRDIGAGK 219

Query: 363 XNYAVNPFPLRDGIDDESYEISIFKPVVEKVIESTFKPNAIVLQCGADSLSGDRLGCFNLNSLK 542
YAVN+PLRDGIDDESYE+IFKPV+ KV+E+F+P+A+VLCGG+DSLSDGRLGCFNL++K
Sbjct: 220 KYAAVNPYPLRDGIDDESYEAIIFKPVVISKVMTFQPSAVVLLQCGSDLSLSDGRLGCFNLTIK 279

Query: 543 GHGKCKVEYMRQQPIPLLMLGGGGYTIRNVARCWTYETALALGTTIPNELPYNDYYEYPT 722
GH KCVE+++ +P+LMLGGGGYTIRNVARCWTYETA+AL T IPNELPYNDY+EYF P
Sbjct: 280 GHAKCVEFVKSFNLPMMLGGGGYTIRNVARCWTYETAVALDTEIPNELPYNDYFEYFPG 339

Query: 723 DFKLHISPSNMANQNTPYEYLERMKQKLFLENRLSIPHAPSVQMIDPEDAMIDDGEGMDN 902
DFKLHISPSNM NCNT EYLE++K+LPENLR +PHAP VQMQ IPEDA+ D G++
Sbjct: 340 DFKLHISPSNMTQNONTNEYLEKKIKQRLFLENLMLPHAPGVQMOPFEDAVQEDSGDE--EE 398

Query: 903 ADPEKRISILASDKYREHEA LDSEDEGD-NRKNVDCKSKR 1028
DP+KRISI SDK + + SDSEDEG+ RKNV FK +
Sbjct: 399 EDPEKRISIRNSDKRISCDEEFSDSEDEGEGRKNVANEKKAK 441

Database: /common/data/swissprot_aa
Posted date: Oct 4, 2009 2:02 AM
Number of letters in database: 157,875,145
Number of sequences in database: 427,028

Lambda K H
0.318 0.134 0.401

Gapped Lambda K H
0.267 0.0410 0.140

Matrix: BLOSUM62
Gap Penalties: Existence: 11, Extension: 1
Number of Hits to DB: 281,587,467
Number of Sequences: 427028
Number of extensions: 557736
Number of successful extensions: 16223
Number of sequences better than 1.0e-10: 1
Number of HSP's better than 0.0 without gapping: 15290
Number of HSP's successfully gapped in prelim test: 0
Number of HSP's that attempted gapping in prelim test: 0
Number of HSP's gapped (non-prelim): 16078
length of database: 157,875,145
effective HSP length: 119
effective length of database: 107,058,813
effective search space used: 30404702892
frameshift window, decay const: 40, 0.1
T: 12
A: 40
X1: 16 (7.3 bits)
X2: 38 (14.6 bits)
X3: 64 (24.7 bits)
S1: 41 (21.7 bits)

Bookmark it!!

See

<http://www.bioperl.org/wiki/HOWTO:SearchIO>

for a GREAT example of a blast report,

code to parse it,

a table of methods,

and the values the methods return.

6

Bio::SearchIO object for BLAST reports

```
#!/usr/bin/perl -w
use strict;
use Bio::SearchIO;
#file: blast_parser_intro.pl

my $blast_report = shift;

my $searchIO_obj = Bio::SearchIO->new(
    -file => $blast_report,
    -format => 'blast'
);
```

6

Result object and methods

```
#file: sample_Blast_parser_1.pl
#!/usr/bin/perl -w
use strict;
use Bio::SearchIO;

my $blast_report = shift;

my $searchIO_obj = Bio::SearchIO->new(
    -file => $blast_report,
    -format => 'blast'
);

while (my $result_obj = $searchIO_obj ->next_result ) {
    my $program = $result_obj ->algorithm;
    my $queryName = $result_obj ->query_name;
    my $queryDesc = $result_obj ->query_description;
    my $queryLen = $result_obj ->query_length;
    print "program=$program\tqueryName=$queryName\n";
    print "queryDesc=$queryDesc\tqueryLen=$queryLen\n";
}
```

Output:

```
program=BLASTX queryName=smed-HDAC1-1 queryDesc=histone deacetylase 1 queryLen=1213 π
```

Object	Method	Example	Description
Result	algorithm	BLASTX	algorithm string
Result	algorithm_version	2.2.4 [Aug-26-2002]	algorithm version
Result	query_name	20521485 dbj AP004641.2	query name
Result	query_accession	AP004641.2	query accession
Result	query_length	3059	query length
Result	query_description	Oryza sativa ... 977CE9AF checksum.	query description
Result	database_name	test.fa	database name
Result	database_letters	1291	number of residues in database
Result	database_entries	5	number of database entries
Result	available_statistics	effectivespaceused ... dbletters	statistics used
Result	available_parameters	gapext matrix allowgaps gapopen	parameters used
Result	num_hits	1	number of hits
Result	hits		List of all <code>Bio::Search::Hit::GenericHit</code> object(s) for this Result
Result	rewind		Reset the internal iterator that dictates where <code>next_hit()</code> is pointing, useful for re-iterating through the list of hits.

71

Hit object and methods

```
#!/usr/bin/perl -w
use strict;
use Bio::SearchIO;
```

```
my $blast_report = shift;

my $searchIO_obj = Bio::SearchIO->new(
    -file => $blast_report,
    -format => 'blast'
);

while (my $result_obj = $searchIO_obj->next_result ) {
    while (my $hit_obj = $result_obj->next_hit){
        my $hitName = $hit_obj->name;
        my $hitAcc = $hit_obj->accession;
        my $hitLen = $hit_obj->length;
        my $hitSig = $hit_obj->significance;
        my $hitScore = $hit_obj->raw_score;

        print "hitName=$hitName\n";
        print "hitAcc=$hitAcc\n";
        print "hitLen=$hitLen\n";
        print "hitSig=$hitSig\n";
        print "hitScore=$hitScore\n";
    }
}
```

must get hit objects
from a result object

Output:

```
hitName=sp|P56517|HDAC1_CHICK hitAcc=P56517 hitLen=480 hitSig=1e-151 hitScore=535
```

72

Hit	name	443893 124775	hit name
Hit	length	331	Length of the Hit sequence
Hit	accession	443893	accession (usually when this is a genbank formatted id this will be an accession number-the part after the <i>gb</i> or <i>emb</i>)
Hit	description	LaForas sequence	hit description
Hit	algorithm	BLASTX	algorithm
Hit	raw_score	92	hit raw score
Hit	significance	2e-022	hit significance
Hit	bits	92.0	hit bits
Hit	hsps		List of all <code>Bio::Search::HSP::GenericHSP</code> object(s) for this Hit
Hit	num_hsps	1	number of HSPs in hit
Hit	locus	124775	locus name
Hit	accession_number	443893	accession number
Hit	rewind		Resets the internal counter for <code>next_hsp()</code> so that the iterator will begin at the beginning of the list

7:

HSP object and methods

#file: sample_Blast_parser.pl

```

#!/usr/bin/perl -w
use strict;
use Bio::SearchIO;

my $blast_report = shift;

my $searchIO_obj = Bio::SearchIO->new(
    -file => $blast_report,
    -format => 'blast'
);

while (my $result_obj = $searchIO_obj->next_result ) {
    while (my $hit_obj = $result_obj->next_hit){
        while (my $hsp_obj = $hit_obj ->next_hsp){
            my $evalue = $hsp_obj->eval;
            my $hitString = $hsp_obj->hit_string;
            my $queryString = $hsp_obj->query_string;
            my $homologyString = $hsp_obj->homology_string;

            print "hsp eval: $evalue\n";
            print "HIT   : ",substr($hitString,0,50)," \n";
            print "HOMOLOGY: ",substr($homologyString,0,50),"\n";
            print "QUERY  : ",substr($queryString,0,50)," \n";
        }
    }
}

```

must get hsp objects
from a hit object



Output:

```

hsp eval: 1e-151
HIT   : CPVFDGLFEFCQLSAGGSVASAVKLNKQQTDIAVNWAGGLHHAKKSEASG
HOMOLOGY: CPVFDGLFEFCQLSAGGSVASAVKLNK + DIA+NW+GGLHHAKKSEASG
QUERY  : CPVFDGLFEFCQLSAGGSVASAVKLNK + DIA+NW+GGLHHAKKSEASG

```

7

<http://www.bioperl.org/wiki/HOWTO:SearchIO>

HSP	algorithm	BLASTX	algorithm
HSP	evalue	2e-022	e-value
HSP	expect	2e-022	alias for evalue()
HSP	frac_identical	0.884615384615385	Fraction identical
HSP	frac_conserved	0.923076923076923	fraction conserved (conservative and identical replacements aka "fraction similar") (only valid for Protein alignments will be same as frac_identical)
HSP	gaps	2	number of gaps
HSP	query_string	DMGRCSSG ..	query string from alignment
HSP	hit_string	DIVQNNS ...	hit string from alignment
HSP	homology_string	D+ SSCCN	string from alignment
HSP	length('total')	52	HSP seq_inds('query','conserved') (966,967,969,971,973,974,975, ...)
HSP	length('hit')	50	HSP seq_inds('hit','identical') (197,202,203,204,205, ...)
HSP	length('query')	15	HSP seq_inds('hit','conserved-not-identical') (198,200)
HSP	hsp_length	52	HSP seq_inds('hit','conserved',1) (197,202-246)
HSP	frame	0	HSP score 227
HSP	num_conserved	48	HSP bits 92.0
HSP	num_identical	46	HSP range('query') (2896,3051)
HSP	rank	1	HSP range('hit') (197,246)
HSP	seq_inds('query','identical')	96	HSP percent_identity 88.4615384615385
HSP	seq_inds('query','conserved-not-identical')	96	HSP strand('hit') 1 HSP strand('query') 1 HSP start('query') 2896 HSP end('query') 3051 HSP start('hit') 197 HSP end('hit') 246 HSP matches('hit') (46,48) HSP matches('query') (46,48) HSP get_align <i>sequence alignment</i> Bio::SimpleAlign object HSP hsp_group <i>Not available in this report</i> HSP links <i>Not available in this report</i>
			links field from WU-BLAST reports run with -topcombo links field from WU-BLAST reports run with -links showing

Other Cool Things

My favorite Bioperl Module, Bio::DB::SeqFeature::Store

Whole set of wrappers for running Bioinformatics tools
in bioperl-run

Run BLAST locally or submit remote jobs (through NCBI)

Run PAML - handles setup and take down of temporary
files and directories

Run alignment progs through similar interfaces: TCoffee, MUSCLE,
Clustalw

Relational Databases for sequence and features

Repository of scripts to do really cool things. (<http://www.bioperl.org/wiki/Scripts>)

Databases

An introduction to using databases for bioinformatics

Jer-Ming Chia (chia@cshl.edu)

What we will do today

1. Creating databases, tables in MySQL
2. Querying and manipulating data using SQL
3. Querying and manipulating data using Perl DBI

What is a database

- A collection of data
 - Text file with a list of genes
 - GFF text file
 - BAM file
 - Excel spreadsheet
 - Set of tables in MySQL

A table of genes

Gene ID	Chromosome	Start	End	Strand	Class
GRMZM2G306328	chr2	175194049	175196453	-1	est
GRMZM2G027393	chr2	175212542	175213269	-1	cdna
GRMZM2G002915	chr2	175243929	175246053	1	est
GRMZM2G419606	chr2	175320426	175321226	-1	cdna
GRMZM2G119906	chr2	175323967	175325504	-1	cdna
GRMZM2G119950	chr2	175325765	175331607	-1	cdna
GRMZM2G125775	chr2	175462240	175463416	-1	cdna
GRMZM2G425965	chr2	175482597	175484512	-1	est
AC195825.3_FG001	chr2	176152209	176155132	-1	fgenesh

- Each row is a *record* of a gene
- Each column is a set of values constrained by a *type*
- A simple query:
What is the location of gene 'GRMZM2G42775'?

A more complex query

Gene table

Gene ID	Chromosome	Start	End	Strand	Class
GRMZM2G306328	chr2	175194049	175196453	-1	est
GRMZM2G027393	chr2	175212542	175213269	-1	cdna
GRMZM2G002915	chr2	175243929	175246053	1	est
GRMZM2G419606	chr2	175320426	175321226	-1	cdna
GRMZM2G119906	chr2	175323967	175325504	-1	cdna
GRMZM2G119950	chr2	175325765	175331607	-1	cdna
GRMZM2G125775	chr2	175462240	175463416	-1	cdna
GRMZM2G425965	chr2	175482597	175484512	-1	est
AC195825.3_FG001	chr2	176152209	176155132	-1	fgenesh

Gene Ontology

Gene ID	Go Term
GRMZM2G002903	nucleic acid binding
GRMZM2G002903	intracellular
GRMZM2G002903	transport
GRMZM2G002915	DNA binding
GRMZM2G002915	transcription factor activity
GRMZM2G002915	nucleus
GRMZM2G002915	transcription
GRMZM2G002915	transcription regulator activity
GRMZM2G002948	multicellular organismal development
GRMZM2G002948	cellular process
GRMZM2G002950	nucleotide binding
GRMZM2G002950	protein binding

Expression data

Gene ID	Exp1	Exp2	Exp3	Exp4
GRMZM2G003109	127.24	86.973	214.73	109.8
GRMZM2G003138	124.73	119.41	125.77	107.08
GRMZM2G003165	77.78	163.4	69.063	51.56
GRMZM2G003167	231.41	420.47	82.018	88.929
GRMZM2G003179	239.6	399.86	483.38	361.11
GRMZM2G003234	107.14	99.023	125.07	84.288
GRMZM2G003246	151.39	94.289	69.389	54.414
GRMZM2G003252	374.61	966.41	560.12	464.19
GRMZM2G003354	4170.1	3378.6	1876.9	2153.5
GRMZM2G003368	13835	5958.7	77.495	100.6

"What are the classes of highly expressed genes in region 50Mb-55Mb of chromosome 5?"

DBMS: Software for managing databases

- Database Management Systems (DBMS)
 - General term for software for managing data
 - Creating tables
 - Loading data
 - Querying data
- E.g: **MySQL, SQLite, Oracle, Microsoft Access, Berkley DB, MongoDB**
- **RDBMS**
 - Relational Database Management System
 - Software for managing related data that is stored across multiple tables

Using a DBMS

1. Through a user-interface
 - E.g: MySQL workbench, HeidiSQL, SequelPro, SQLite Manager, SQLite Spy
2. Programmatically through SQL
 - Structured Query Language
 - E.g: "select gene_id from gene_table where chromosome = 'chr2';"
3. Programmatically through an API in another programming language
 - Perl DBI
 - Java JDBC, C ODBC

MySQL

- A robust RDBMS is very popular for large bioinformatics databases. Great for:
 - Very large, persistent datasets
 - Multi users with different permission levels
 - High volume transactions
- To access the mysql client from the command line, you need 4 pieces of information:
 1. Host (Defaults to localhost)
 2. Port (Defaults to 3306)
 3. Username
 4. Password

When MySQL is first installed a 'root' account, without a password, is created by default. We will use this account in exercises.

Using MySQL Shell

- Start the MySQL client from the command line, and this will bring up a MySQL shell, connected to the MySQL server on your local machine:

```
$ mysql -u root
```

- For example, to connect to the public Ensembl MySQL:

```
$ mysql -h ensembldb.ensembl.org -P 5306 -u anonymous
```

- Type commands into the MySQL shell, each line must terminate with a ';

```
mysql> show databases;  
mysql> create database myTestDB; # myTestDB is the database name  
mysql> use myTestDB; # Use database myTestDB  
mysql> help;
```

- To quit:

```
mysql> \q;
```

- To cancel a command:

```
mysql> \c;
```

Creating a table

- Things to consider:
 - Table name
 - Name of each column
 - Data type of each column
 - Range of values of data in each column

Basic Datatypes

- Numeric
 - INT : for integers
 - Double : numerical data with decimals
- Strings
 - CHAR : for strings up to 255 in length
 - TEXT : large strings
- Lots more on the MySQL website
<http://dev.mysql.com/doc/refman/5.6/en/data-types.html>
- And also, a cheat sheet of course.
<http://en.wikibooks.org/wiki/MySQL/CheatSheet>

SQL: Creating a table

Syntax:

```
CREATE TABLE tablename (
    column_1_name datatype [optional constraint]
    column_2_name datatype [optional constraint]
    ...
);
```

For example, to create this table using SQL:

gene_id	chr	start	end
GRMZM2G306328	chr2	175194049	175196453
GRMZM2G027393	chr2	175212542	175213269
GRMZM2G002915	chr2	175243929	175246053
GRMZM2G419606	chr2	175320426	175321226

```
CREATE TABLE genes (
    gene_id` char(25) NOT NULL DEFAULT '',
    `chr` char(5) NOT NULL DEFAULT '',
    `start` int(9) NOT NULL DEFAULT '0',
    `end` int(9) NOT NULL DEFAULT '0',
    PRIMARY KEY (`gene_id`)
);
```

Keys and Indexes

- INDEX
 - Synonymous with KEY
 - It is the lookup column, or a set of columns, for a table.
 - There can be more than one KEY in a table
- PRIMARY KEY
 - The primary key for a table represents the column, or set of columns, that is mostly frequently used as an index to the table
 - Columns used as the primary keys must be contain values that are unique to each row
 - There can only be one primary key in a table

SQL : Simple Queries

- Query using SELECT

```
mysql> use progbio2012; # Use database progbio2012
mysql> SELECT gene_id from genes; #SELECT Column [columns] from Table
```

- Use LIMIT when the list is too long

```
SELECT gene_id, chr, start, end FROM genes limit 10;
```

- Wildcard character “*” for all columns in table

```
SELECT * from genes limit 10;
```

SQL: SELECT ... WHERE for filtering results

what are the genes on chromosome 5?

```
SELECT gene_id FROM genes WHERE chr='chr5';
```

what are the genes that lie within 50Mb – 55 Mb of chromosome 5?

```
SELECT gene_id FROM genes  
WHERE chr='chr5' and end >= 50000000 and start <= 55000000;
```

SQL: SELECT ... WHERE with OR

what are the genes that lie within chr5 with evidence 'est' or 'cdna'?

```
SELECT gene_id , class  
FROM genes  
WHERE chr='chr5'  
and (evidence='cdna' OR evidence= 'est');
```

SQL: Sorting and Distinct

What are the last 20 genes on chr 10?

```
SELECT gene_id, chr, start, end FROM genes  
WHERE chr='chr10'  
ORDER BY end desc  
LIMIT 20;
```

What is the unique list of gene evidences in the genes table?

```
SELECT DISTINCT evidence FROM genes;
```

SQL: SELECT COUNT... GROUP BY

How many rows are there in the table?

```
SELECT COUNT(*) FROM genes;
```

How many genes lie in chromosome 5?

```
SELECT COUNT(*) FROM genes WHERE chr='chr5';
```

How many genes are there in each chromosome?

```
SELECT chr, COUNT(*) FROM genes GROUP BY chr;
```

Other SQL functions besides COUNT:

- avg, min, max, concat

SQL: Select ... Join

Gene table

Gene ID	Chromosome	Start	End	Strand	Class
GRMZM2G306328	chr2	175194049	175196453	-1	est
GRMZM2G027393	chr2	175212542	175213269	-1	cdna
GRMZM2G002915	chr2	175243929	175246053	1	est
GRMZM2G419606	chr2	175320426	175321226	-1	cdna
GRMZM2G119906	chr2	175323967	175325504	-1	cdna
GRMZM2G119950	chr2	175325765	175331607	-1	cdna
GRMZM2G125775	chr2	175462240	175463416	-1	cdna
GRMZM2G425965	chr2	175482597	175484512	-1	est
AC195825_3_FG001	chr2	176152209	176155132	-1	fgenesh

Expression data

Gene ID	Exp1	Exp2	Exp3	Exp4
GRMZM2G003109	127.24	86.973	214.73	109.8
GRMZM2G003138	124.73	119.41	125.77	107.08
GRMZM2G003165	77.78	163.4	69.063	51.56
GRMZM2G003167	231.41	420.47	82.018	88.929
GRMZM2G003179	239.6	399.86	483.38	361.11
GRMZM2G003234	107.14	99.023	125.07	84.288
GRMZM2G003246	151.39	94.289	69.389	54.414
GRMZM2G003252	374.61	966.41	560.12	464.19
GRMZM2G003354	4170.1	3378.6	1876.9	2153.5
GRMZM2G003368	13835	5958.7	77.495	100.6

Gene Ontology

Gene ID	Go Term
GRMZM2G002903	nucleic acid binding
GRMZM2G002903	intracellular
GRMZM2G002903	transport
GRMZM2G002915	DNA binding
GRMZM2G002915	transcription factor activity
GRMZM2G002915	nucleus
GRMZM2G002915	transcription
GRMZM2G002915	transcription regulator activity
GRMZM2G002948	multicellular organismal development
GRMZM2G002948	cellular process
GRMZM2G002950	nucleotide binding
GRMZM2G002950	protein binding

SQL :Simple JOIN

What are the expression values for all transcription factors (Gene ontology id= GO:0030528) in experiment 1?

```
SELECT genes_go.gene_id, go_id, day1
FROM genes_go, expression
WHERE genes_go.gene_id = expression.gene_id
and go_id = 'GO:0030528';
```

Let's try this

*"What are the classes of highly expressed genes in region
50Mb–55Mb of chromosome 5?"*

Perl DBI

- DBI is a module that provides access to DBMS in Perl
- It hides the nuts and bolts for connecting to each type of DBMS, leaving a consistent interface for connecting to a database
- The key object in DBI is the database handle (\$dbh), which represents a connection to a DBMS.

Perl DBI: 3 easy steps

1. Create a database handle
`$dbh = DBI->connect(...)`
2. Execute a SQL statement using the database handle
`$dbh->do something`
3. Disconnect the handle
`$dbh->disconnect`

Perl DBI Step 1: Constructing the handle

- for MySQL:

```
my $dbname = 'prog2011';
my $driver = 'mysql';
my $user = 'root';
my $passwd = '';
my $host = 'localhost';
my $port = 3306;

my $dsn = "DBI:$driver:database=$dbname;host=$host;port=$port";
my $dbh = DBI->connect($dsn,$user,$passwd);
```

- for SQLite:

```
my $dbname = 'prog2011';
my $driver='SQLite';
my $dsn = "DBI:$driver:$dbname";
my $dbh = DBI->connect($dsn);
```

Perl DBI Step 2: Executing the SQL

1. Construct the SQL query

```
my $sql = "SELECT count(*) From gene";
```

2. Execute the transaction using the database handle

Querying and fetching data:

```
my $results_array_ref = $dbh->selectall_arrayref($sql);
```

DBI Example: Executing an SQL

For example, to create a table using
`$dbh->do()`

```
=pseudocode
$dbh = DBI->connect($dsn)
$dbh->do(SQL)
$dbh->disconnect
=end
```

```
#!/usr/bin/perl
use strict;
use warnings;
use DBI;

my $dbname = 'progbio2012';
my $user = 'root';
my $passwd = '';
my $host = 'localhost';
my $port = 3306;

my $dsn = "DBI:mysql:database=$dbname;host=$host;port=$port";
my $dbh = DBI->connect($dsn,$user,$passwd);

my $sql = "CREATE table foo (bar char(10));" ;
$dbh->do($sql);

$dbh->disconnect;
exit;
```

DBI : Fetch a list of genes using DBI

`$dbh->selectall_arrayref`

```
=pseudocode
$dbh = DBI->connect($dsn)
$fetched_results1 = $dbh->fetch SQL query
do something with results
$dbh->disconnect
=end
```

```
#!/usr/bin/perl
use strict;
use warnings;
use DBI;

my $dbname = 'progbio2012';
my $user = 'root';
my $passwd = '';
my $host = 'localhost';
my $port = 3306;

my $dsn = "DBI:mysql:database=$dbname;host=$host;port=$port";
my $dbh = DBI->connect($dsn,$user,$passwd);

my $query = "SELECT gene_id, chr, start, end from genes";
my @results = @{$dbh->selectall_arrayref($query)};

foreach my $row_ref (@results){
    my $str = join("\t", @{$row_ref});
    print $str, "\n";
}

$dbh->disconnect;
exit;
```

DBI : Count using selectrow_arrayref

For SQL queries which will only return
a single row,
e.g: SELECT COUNT query

`$dbh->selectrow_arrayref`

```
=pseudocode
$dbh = DBI->connect($dsn)
$fetched_row = $dbh->fetch SQL query
do something with results
$dbh->disconnect
=end
```

```
#!/usr/bin/perl
use strict;
use warnings;
use DBI;

my $dbname = 'progbio2012';
my $user = 'root';
my $passwd = '';
my $host = 'localhost';
my $port = 3306;

my $dsn = "DBI:mysql:database=$dbname;host=$host;port=$port";
my $dbh = DBI->connect($dsn,$user,$passwd);

my $query = "SELECT COUNT(*) from genes where chr ='chr10'";
my @row = @{$dbh->selectrow_arrayref($query)};
print join ("\t", @row), "\n";

$dbh->disconnect;
exit;
```

DBI : Querying using placeholders

Fetch the expression level for a list of genes

```
=pseudocode
$dbh = DBI->connect($dsn)
$stmt = $dbh->prepare(SQL)
loop:
    $stmt->execute
    $stmt->fetchrow_array;
end loop
$dbh->disconnect
=end
```

```
my $dbh = DBI->connect( $dsn, $user, $passwd );
my $query = "SELECT day1, day2, day3, day4
            FROM expression
            WHERE gene_id = ?";
my $sth = $dbh->prepare($query);

my $file = shift;
open IN,<,$file || die ("Can't open file $file $!");
while (my $gene_id = ){
    chomp $gene_id;
    $sth->execute($gene_id);
    my @results = @{$sth->fetchall_arrayref};
    foreach my $row_ref (@results){
        my $str = join "\t", @{$row_ref};
        print $gene_id, "\t", $str, "\n";
    }
}
close IN;
$dbh->disconnect;
exit;
```

DBI: Placeholders vs SelectAll

Placeholders:

```
=pseudocode
@geneList = read from file
$dbh = DBI->connect($dsn)

$sql = "SELECT day1
        FROM expression
        WHERE gene_id = ?"

$stmt = $dbh->prepare($sql)

loop through geneList:
    $stmt->execute($gene)
    $expr = $stmt->fetchrow_array
    print $expr
end loop

$dbh->disconnect
=end
```

- 1 database transaction for each gene queried
- Slow if you have many genes to query

SelectAll:

```
=pseudocode
@geneList = read from file
$dbh = DBI->connect($dsn)

$sql = "SELECT gene, day1
        FROM expression";

%gene_expr hash

for each result in
    $db->selectall_arrayref($sql):
        $gene_expr hash{$gene} = $expr
    end for each result

loop through geneList:
    if exist in %gene_expr hash
        print $expr
    end loop

$db->disconnect
=end
```

- A single database transaction
- Slow if you're fetching millions of rows and you end up only needing a small fraction

For other DBI functions, see CPAN

With \$dbh->selectall_hashref

```
my $dsn = "DBI:mysql:database=$dbname;host=$host;port=$port";
my $dbh = DBI->connect($dsn,$user,$passwd);
my $query = "SELECT gene_id, chr, start, end from genes";
my %results = %{$dbh->selectall_hashref($query,'gene_id')};
foreach my $gene (keys %results){
    print $gene, "\t";
    $results{$gene}->{ 'chr' }, "\t",
    $results{$gene}->{ 'start' }, "\t",
    $results{$gene}->{ 'end' }, "\n";
}
```

Other useful MySQL commands

For loading a text file into a table from the unix command line:

```
$ mysqlimport --local -u root databasename filename.txt
```

Filename must have the same name as the table you are trying to load data into
For dumping data from a table:

```
$ mysqldump -u root databasename tablename > table.sql
```

Or for dumping the entire MySQL database:

```
$ mysqldump -u root databasename > database.sql
```

Other useful SQL commands

Inserting new rows into table

```
INSERT into genes (gene_id, chr, start, end, evidence)
values ('foo','chrX',1000,1500,'cdna');
```

Updating data in table

```
UPDATE genes SET gene_id = 'bar' WHERE gene_id = 'foo';
```

Before you leap:

- What we did not cover and should be considered before charging ahead and designing your own databases
 - The full JOIN Syntax:

```
mysql> SELECT genes_go.gene_id, go_id, day1
      from genes_go
      join ( expression )
    on ( genes_go.gene_id = expression.gene_id )
   where go_id = 'GO:U030528';
```

- Database Normalization
- Consider other RDMBS: e.g SQLite

Problem sets: Databases and Perl DBI

- For this problem set, we have installed MySQL servers in each of the machines.
- In each MySQL server, you'll find a database named progbio2012.
- This database has 5 tables: genes, genes_go, expression,.snp, go_terms
 - 1. 'genes' table lists the location and evidence class of each gene
 - 2. 'genes_go' table contains the Gene Ontology terms for genes that have a Gene Ontology annotation
 - 3. 'go_terms' list the Gene Ontology descriptions of the GO IDs
 - 4. 'expression' table contains the expression level of genes in 4 experiments.
 - 5. 'snps' list of SNPs in chr1 and chr10.
- You will need the data in these tables for the problem sets

Problem sets: Getting familiar with MySQL

1. Follow the steps below to enter the MySQL shell and use the database progbio2012.

- on the unix command line:

```
$ mysql -u root
```

You're now in the MySQL shell.

- To use the database progbio2012:

```
mysql> use progbio2012;
```

To list the tables in the database:

```
mysql> show tables;
```

Problem sets: Getting familiar with mySQL

2. Use the “explain” command to see the schema of each table

- on the unix command line:

```
mysql> explain genes;
```

- Try out SHOW command to see the SQL-CREATE syntax for each table:

```
mysql> show create table genes;
```

Problem sets: SQL

3. Using SQL, perform the following queries:

- a. How many rows are there in the gene table?

- b. How many genes have GO annotations?

HINT: count(distinct gene_id)

- c. List the number of genes in each evidence class in the genes table

HINT: Using a COUNT...GROUP BY query

- d. How many genes in the first 100Mb of chr1 contain snps?

- e. Using a SQL query that joins the genes_go and expression table,
select the day1 value of genes that have the go_term ‘chromatin binding’;

- f. Try the query above again, but limit it to genes on chr1.

Problem sets: Perl DBI

4. Do the following using Perl-DBI

- Using a similar query to Q3.e above, write a Perl DBI script that produces a tab-delimited text file for genes with go_term 'nucleic acid binding'.

The text file should have the columns: gene_id, go_term, day1, day2, day3, day4

- Construct a query to find genes where the expression level in day4 is greater than day1.

Print out this list of genes.

5. Advanced DBI problems

- Compute the gene density on chr10 across 1Mb windows.

Assume that Chr10 has a total length of 150Mb.

- Compute the average expression level in each experiment (day1, day2, day3, day4) for the genes in each GO term.

HTML

HTML

- HyperText Markup Language
- Not a programming language
- Stored in text files (just like Perl)

A basic page

```
<html>
  <head>
    <title>My web page title</title>
  </head>
  <body>
    Your HTML content here
  </body>
</html>
```

A kosher page

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

  <head>
    <title>An XHTML 1.0 Strict standard template</title>
  </head>
  <body>
    <p>... Your HTML content here ...</p>
  </body>
</html>
```

Why use web Standards?

- Accessibility
 - To robots
 - To people
- Stability

<Tags />

Most tags open and close

- Tags must be nested properly

Right	<pre> Strong and emphasis </pre>	Wrong	<pre> Strong and emphasis </pre>
--------------	--	--------------	--

- Some tags stand alone

```
<br /> <hr />
```

- Some tags take attributes

```

```

```
<a href="theonion.com">The Onion</a>
```

- Elements consist of start and end tags flanking content

XHTML tags

<http://www.w3schools.com/tags/>

Text tags

- Heading tag

```
<H1>This is a top level heading</H1>  
<H6>This is the bottom level heading</H6>
```

- Paragraph tag

```
<p>This is definitely a paragraph</p>
```

- Line Break

This is just two lines
`
` With a hard break.

- Emphasis and Strong

That's `exactly` what I mean - I am `sick` of this slide

- Comment Tag

```
<!-- This is a comment. You won't see this on the web-->
```

Tables

```
<table border="1">

<tr>
<th> Column 1 heading</th>
<th> Column 2 heading</th>
<th> Column 3 heading</th>
</tr>

<tr>
<td>Row 2, cell 1</td>
<td colspan="2">Row 2, cell 2, also spanning Row 2, cell 3</td>
</tr>

<tr>
<td rowspan="2">Row 3, cell 1, also spanning Row 4, cell 1</td>
<td>Row 3, cell 2</td>
<td>Row 3, cell 3</td>
</tr>

<tr>
<td>Row 4, cell 2</td>
<td>Row 4, cell 3</td>
</tr>

</table>
```

output:

Column 1 heading	Column 2 heading	Column 3 heading
Row 2, cell 1	Row 2, cell 2, also spanning Row 2, cell 3	
Row 3, cell 1, also spanning Row 4, cell 1	Row 3, cell 2	Row 3, cell 3

<http://htmldog.com/guides/htmlintermediate/tables/>

Lists

```
<ol>
<li>First things first</li>
<ul>
<li>Who you know</li>
</ul>
<li>Not</li>
<ul>
<li>What you know</li>
<li>What you can do with it</li>
</ul>
</ol>
```

output:

1. First things first
 - Who you know
2. Not
 - What you know
 - What you can do with it

Links

- Relative

```
<a href="myDirectory/index.html">Go down a directory</a>
<a href="..../index.html">Go up a directory</a>
```

- Absolute

```
<a href="/">Go to the root</a>
<a href="http://nytimes.com">Go to the NY Times</a>
```

- Anchors

```
<a href="#theEnd">Go to the end</a>
<h1 id="theEnd">This is the end</h1>
```

Images

```

```



Forms

```
<form name="input" action="html_form_submit.pl" method="post">
```

- POST vs GET

GET = Data is in the URL

POST = Data is in the message body

Text Fields

```
<form name="input" action="handleMyForm.pl" method="get">
  First name:
  <input type="text" name="firstname" />
  <br/>
  Last name:
  <input type="text" name="lastname" />
  <input type="submit" value="Submit" />
</form>
```

output:

First name:

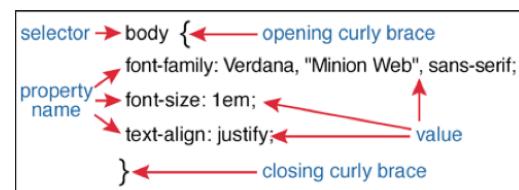
Last name:

Cascading Style Sheets

- Help separate **content** from **appearance**
 - One style sheet can be applied to hundreds of web pages
 - Change styles in just one location

How CSS works

- Statements consist of
 - Selectors
 - Declarations
 - Properties:Values (units)



CSS:Where do I put it?

- Embedded in the `<head>` of each page

```
<head><style type="text/css"> </style></head>
```

- Linked in the `<head>`

Advantages: templating, speed

```
<link rel="stylesheet" type="text/css"
      href="/styles/style.css" />
```

- Inline (avoid this)

```
<p style="color: red">text</p>
```

CSS Selectors

- HTML selectors - raw tags in the style sheet)

- Class selectors

use .className in style sheet

use class="className" in HTML

- ID selectors

use #idName in style sheet

use id="idName" in HTML

Divs and Spans

- Divs

- Use <div id="myDiv"> </div> to define block elements. Useful for both formatting and positioning.
- The id is unique. It refers to one element

- Spans

- Use when you want to apply a class to some text inline
- This is my sequence

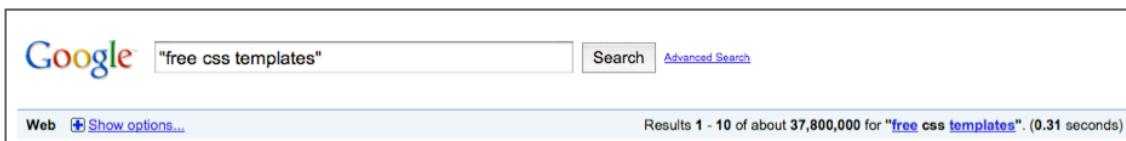
```
<span class="dna">ACTGATCTAGCT</span>
```

BlueprintCSS

CSS framework

- grid
- “sensible typography”
- stylesheet for printing

Do Not Reinvent the Wheel



Results 1 - 10 of about 317,000 for "two column css". (0.40 seconds)

<http://www.freecsstemplates.org>

Where does my website go?

- On Mac OS X
 - Personal web: [~/Sites](#)
 - Main web: [/Library/Webserver/Documents](#)
- Linux: [/var/www/html](#) or [/var/apache2/htdocs](#)
- XP Home: [C:\Program Files\ApacheGroup\Apache\htdocs](#)
- Could be elsewhere. Don't give up!

Naming your html files

- .html .htm
- Why index.html is special

Resource: HTML

- HTML Dog

<http://htmldog.com>

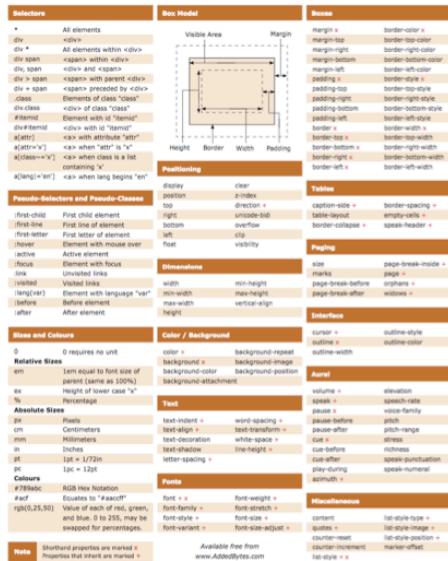


- W3C tags

<http://www.w3schools.com/tags>



Resources: CSS



Cheat sheet:

<http://www.addedbytes.com/download/css-cheat-sheet-v2/pdf/>

CSS tutorial

http://westciv.com/wiki/Main_Page

Two column style sheet and tutorial

http://www.456bereastreet.com/lab/developing_with_web_standards/csslayout/2-col/

Tools of the Trade

- Web Developer Plugin for Firefox
- CSS editors
 - MacRabbit CSSEdit
 - SimpleCSS
 - TopStyle (Windows)

NGS Sequence data

Jason Stajich

UC Riverside

jason.stajich[at]ucr.edu

twitter:[hyphaltip](#) [stajichlab](#)

Lecture available at http://github.com/hyphaltip/CSHL_2012_NGS

NGS sequence data

- Quality control
- Alignment
- Variant calling
 - SNPs
 - Indels

Sequence data sources

- Sanger
 - Long reads, high quality, expensive
- Illumina
 - Short reads 50–150bp (HiSeq) and up to 250bp (MiSeq)
 - Cheap and Dense read total (HiSeq 200–300M paired-reads for ~\$2k)
- 454
 - Longish reads 300–500 bp, some homopolymer seq problems,
 - Expensive (\$10k for 1M reads), recent chemistry problems
- PacBio
 - Long reads, but small amount (10k)
 - Low seq quality and not cheap
 - Can help augment assemblies, but not good enough on its own

Sequence data source (cont)

- SOLiD
 - Short reads, 30–50bp. Reasonably price-point for the density
 - 1/5 as many reads as Illumina HiSeq
- Ion Torrent
 - Cheaper machine, fast, 100bp reads and reported 100M
 - Quality okay for some applications

Sequencer comparisons

Glenn TC, "Field guide to next-generation DNA sequencers" DOI:[10.1111/j.1755-0998.2011.03024.x](https://doi.org/10.1111/j.1755-0998.2011.03024.x)

Table 2 Comparison of sequencing instruments, sorted by cost/Mb, with expected performance by mid 2011

Instrument	Run time ^a	Millions of reads/run	Bases/read ^b	Yield Mb/run	Reagent cost/run ^c	Reagent cost/Mb	Minimum unit cost (% run) ^d
3730xl (capillary)	2 h	0.000096	650	0.06	\$96	\$1500	\$6 (1%)
Ion Torrent – '314' chip	2 h	0.10	100	>10	\$500	<\$50	~\$750 (100%)
454 GS Jr. Titanium	10 h	0.10	400	50	\$1100	\$22	\$1500 (100%)
Starlight*	†	~0.01	>1000	†	†	†	†
PacBio RS	0.5–2 h	0.01	860–1100	5–10	\$110–900	\$11–180	†
454 FLX Titanium	10 h	1	400	500	\$6200	\$12.4	\$2000 (10%)
454 FLX+ ^e	18–20 h	1	700	900	\$6200	\$7	\$2000 (10%)
Ion Torrent – '316' chip*	2 h	1	>100	>100	\$750	<\$7.5	~\$1000 (100%)
Helicos ^f	N/A	800	35	28 000	N/A	NA	\$1100 (2%)
Ion Torrent – '318' chip*	2 h	4–8	>100	>1000	~\$925	~\$0.93	~\$1200 (100%)
Illumina MiSeq*	26 h	3.4	150 + 150	1020	\$750	\$0.74	~\$1000 (100%)
Illumina iScanSQ	8 days	250	100 + 100	50 000	\$10 220	\$0.20	\$3000 (14%)
Illumina GAIIX	14 days	320	150 + 150	96 000	\$11 524	\$0.12	\$3200 (14%)
SOLiD – 4	12 days	>840 ^g	50 + 35	71 400	\$8128	<\$0.11	\$2500 (12%)
Illumina HiSeq 1000	8 days	500	100 + 100	100 000	\$10 220	\$0.10	\$3000 (12%)
Illumina HiSeq 2000	8 days	1000	100 + 100	200 000	\$20 120 ^h	\$0.10	\$3000 (6%)
SOLiD – 5500 (PI)*	8 days	>700 ^g	75 + 35	77 000	\$6101	<\$0.08	\$2000 (12%)
SOLiD – 5500xl (4hq)*	8 days	>1410 ^g	75 + 35	155 100	\$10 503 ^h	<\$0.07	\$2000 (12%)
Illumina HiSeq 2000 – v3 ⁱ *	10 days	≤3000	100 + 100	≤600 000	\$23 470 ^h	≥\$0.04	~\$3500 (6%)

File formats

FASTQ

```
@SRR527545.1 1 length=76
GTCGATGATGCCTGCTAAACTGCAGCTTACGTACTGCGGACCCCTGCAGTCAGCGCTCGTCATGGAACGCAAACG
+
HHHHHHHHHHHHFGHHHHHHFHHGHGGHEEEHHHHEFFHHHFHHHBHHHEFH?CEDCBFEFFFFAFDF9
```

FASTA format

```
>SRR527545.1 1 length=76
GTCGATGATGCCTGCTAAACTGCAGCTTACGTACTGCGGACCCCTGCAGTCAGCGCTCGTCATGGAACGCAAACG
```

SFF – Standard Flowgram Format – binary format for 454 reads

Colorspace (SOLiD) – CSFASTQ

```
@0711.1 2_34_121_F3
T1133232100221013101113133220002000120000200001000
+
64;;9:;>+0*&.*1-.5($2$3&$570*$575&$9966$5835'665
```

Quality Scores in FASTQ files

S - Sanger Phred+33, raw reads typically (0, 40)
X - Solexa Solexa+64, raw reads typically (-5, 40)
I - Illumina 1.3+ Phred+64, raw reads typically (0, 40)
J - Illumina 1.5+ Phred+64, raw reads typically (3, 40)
with 0=unused, 1=unused, 2=Read Segment Quality Control Indicator (**bold**)
(Note: See discussion above).

Read naming

ID is usually the machine ID followed by flowcell number column, row, cell of the read.

Paired-End naming can exist because data are in two file, first read in file 1 is paired with first read in file 2, etc. This is how data come from the sequence base calling pipeline. The trailing /1 and /2 indicate they are the read-pair 1 or 2.

In this case #CTTGTA indicates the barcode sequence since this was part of a multiplexed run.

File: Project1_lane6_1_sequence.txt

```
@HWI-ST397_0000:2:1:2248:2126#CTTGTA/1
TTGGATCTGAAAGATGAATGTGAGAGACACAATCCAAGTCATCTCTCATG
+HWI-ST397_0000:2:1:2248:2126#CTTGTA/1
eeee\dZddadddddeeeeeedaed_ec_ab_\NSRN Rcd ddc[_c^d
```

File: Project1_lane6_2_sequence.txt

```
@HWI-ST397_0000:2:1:2248:2126#CTTGTA/2
CTGGCATTTCACCCAAATTGCTTTAACCTGGGATCGTATTACCAA
+HWI-ST397_0000:2:1:2248:2126#CTTGTA/2
]YYY_\[\da_da_aa_a_a_b_Y]Z]ZS[\L[\ddccbdYc\ecacX
```

Paired-end reads

These files can be interleaved, several simple tools exist, see velvet package for shuffleSequences scripts which can interleave them for you.

Interleaved was required for some assemblers, but now many support keeping them separate. However the order of the reads must be the same for the pairing to work since many tools ignore the IDs (since this requires additional memory to track these) and instead assume in same order in both files.

Orientation of the reads depends on the library type. Whether they are

----> <---- Paired End (Forward Reverse)
<---- ----> Mate Pair (Reverse Forward)

Data QC

- Trimming
 - FASTX_toolkit, sickle
 - Adapative or hard cutoff
- Additional considerations for Paired-end data
- Evaluating quality info with reports

FASTX toolkit

- Useful for trimming, converting and filtering FASTQ and FASTA data
- One gotcha – Illumina quality score changes from 64 to 33 offset
- Default offset is 64, so to read with offset 33 data you need to use -Q 33 option
- fastx_quality_trimmer
- fastx_splitter – to split out barcodes
- fastq_quality_formatter – reformat quality scores (from 33 to 64 or)
- fastq_to_fasta – to strip off quality and return a fasta file
- fastx_collapse – to collapse identical reads. Header includes count of number in the bin

FASTX – fastx_quality_trimmer

- Filter so that X% of the reads have quality of at least quality of N
- Trim reads by quality from the end so that low quality bases are removed (since that is where errors tend to be)
- Typically we use Phred of 20 as a cutoff and 70% of the read, but you may want other settings
- This is adaptive trimming as it starts from end and removes bases
- Can also require a minimum length read after the trimming is complete

FASTX toolkit – fastx_trimmer

- Hard cutoff in length is sometimes better
- Sometimes genome assembly behaves better if last 10–15% of reads are trimmed off
- Adaptive quality trimming doesn't always pick up the low quality bases
- With MiSeq 250 bp reads, but last 25–30 often low quality and HiSeq with 150 bp often last 20–30 not good quality
- Removing this potential noise can help the assembler perform better

Trimming paired data

- When trimming and filtering data that is paired, we want the data to remain paired.
- This means when removing one sequence from a paired-file, store the other in a separate file
- When finished will have new File_1 and File_2 (filtered & trimmed) and a separate file File_unpaired.
- Usually so much data, not a bad thing to have aggressive filtering

Trimming adaptors

- A little more tricky, for smallRNA data will have an adaptor on 3' end (usually)
- To trim needs to be a matched against the adaptor library – some nuances to make this work for all cases.
 - What if adaptor has low quality base? Indel? Must be able to tolerate mismatch
- Important to get right as the length of the smallRNAs will be calculated from these data
- Similar approach to matching for vector sequence so a library of adaptors and vector could be used to match against
- Sometimes will have adaptors in genomic NGS sequence if the library prep did not have a tight size distribution.

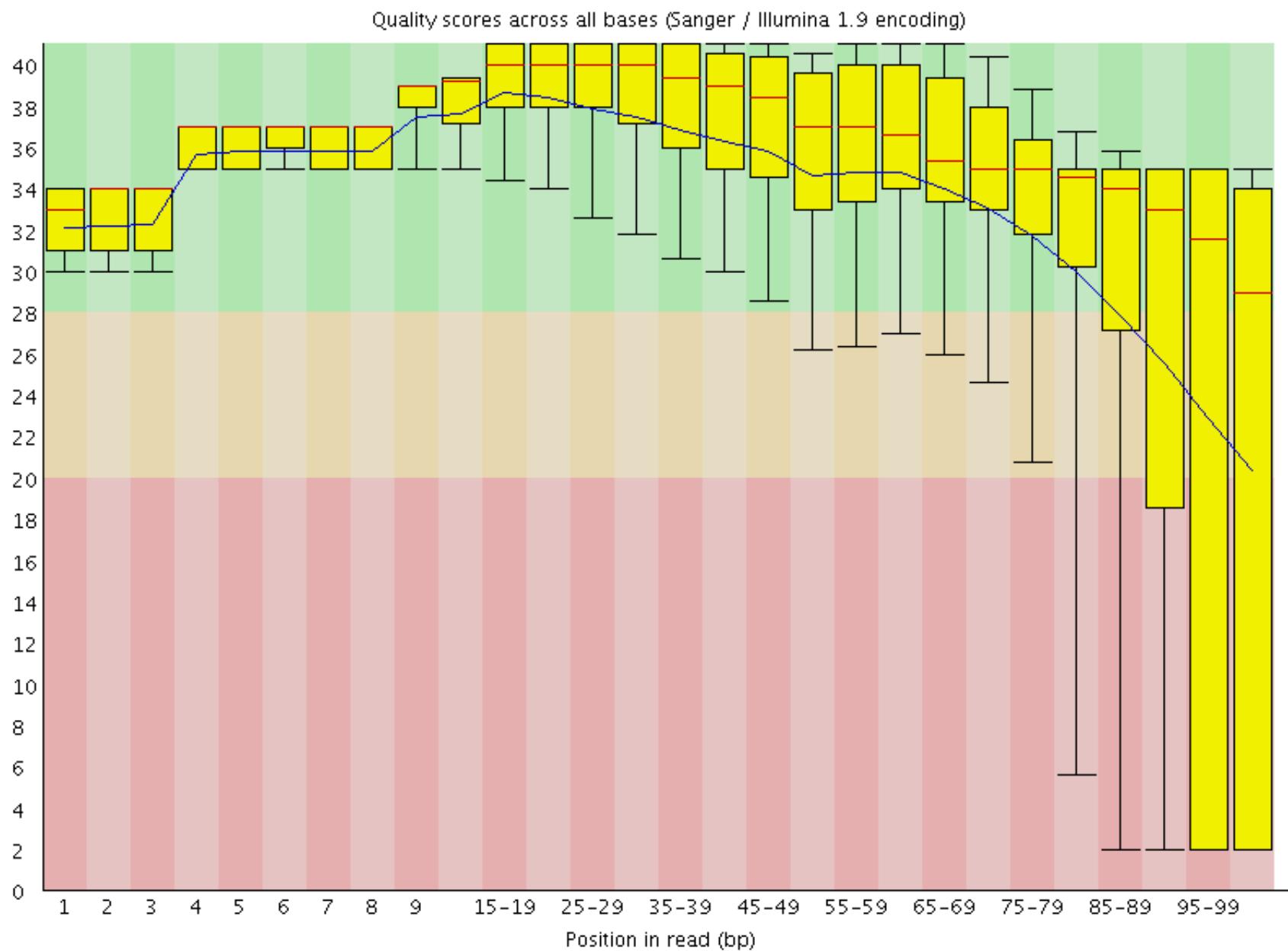
Trimming adaptors – tools

- cutadapt – Too to matching with alignment. Can search with multiple adaptors but is pipelining each one so will take 5X as long if you match for 5 adaptors.
- SeqPrep – Preserves paired-end data and also quality filtering along with adaptor matching

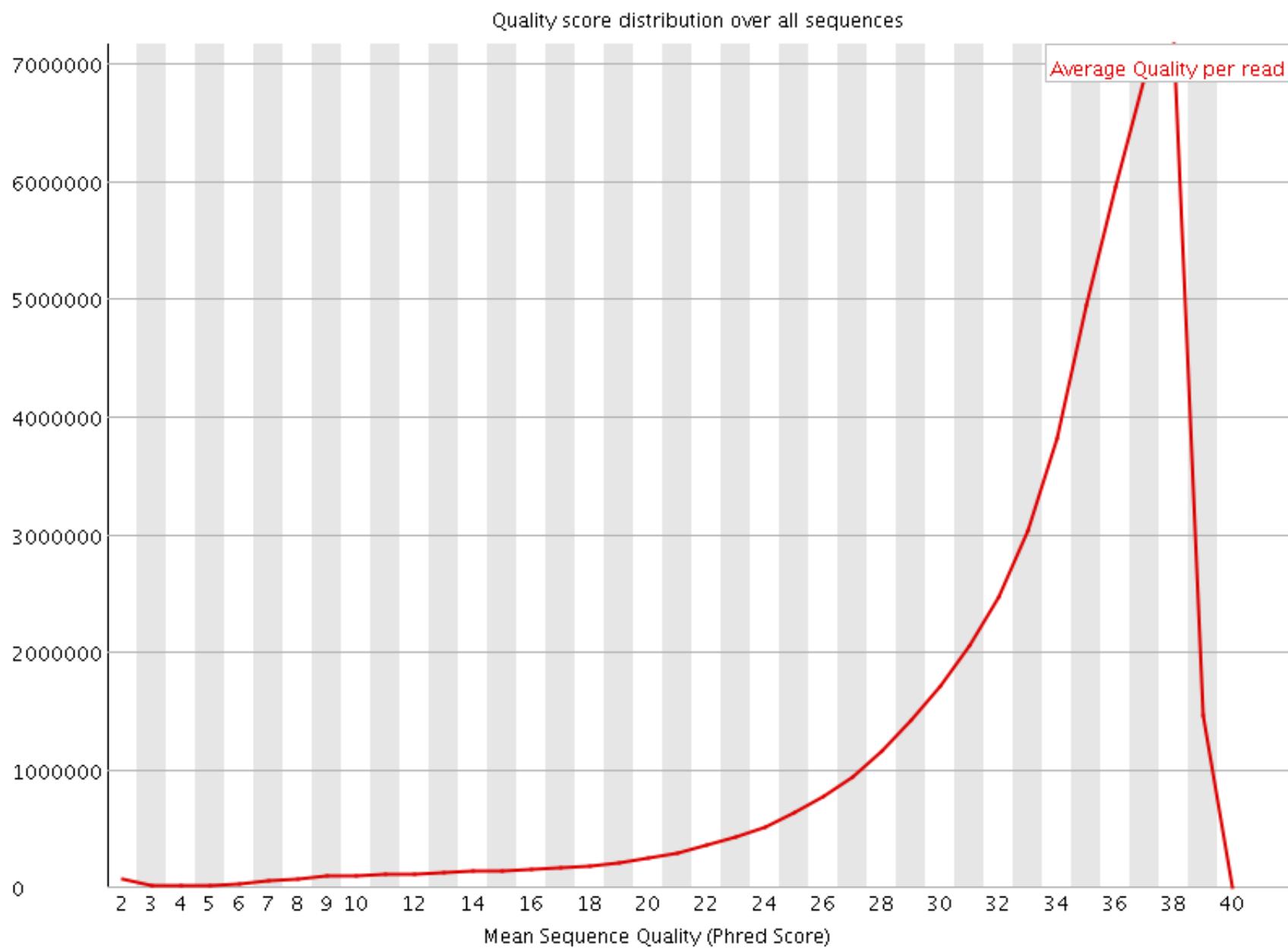
FASTQC for quality control

- Looking at distribution of quality scores across all sequences helpful to judge quality of run
- Overrepresented Kmers also helpful to examine for bias in sequence
- Overrepresented sequences can often identify untrimmed primers/adaptors

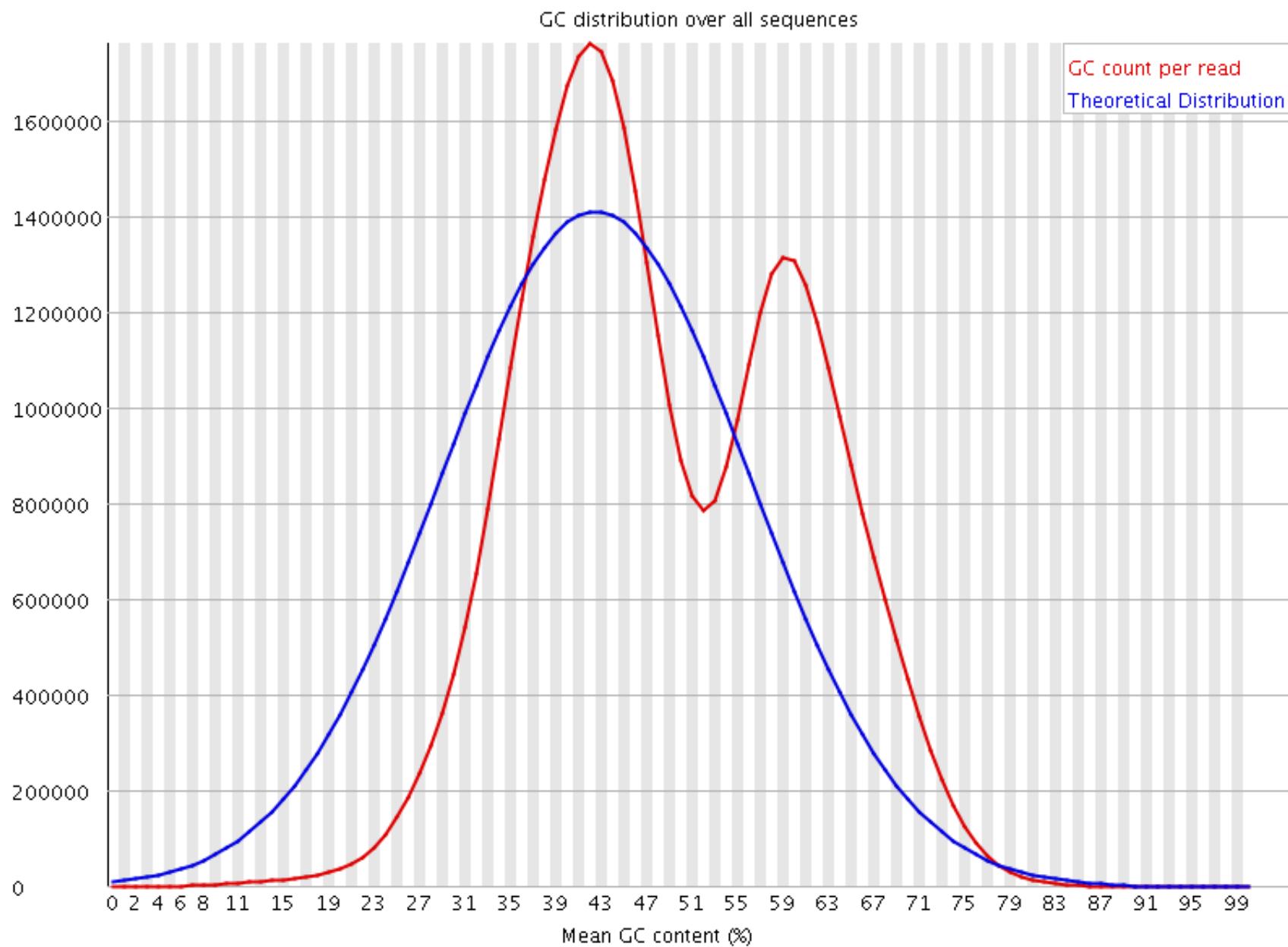
FASTQC - per base quality



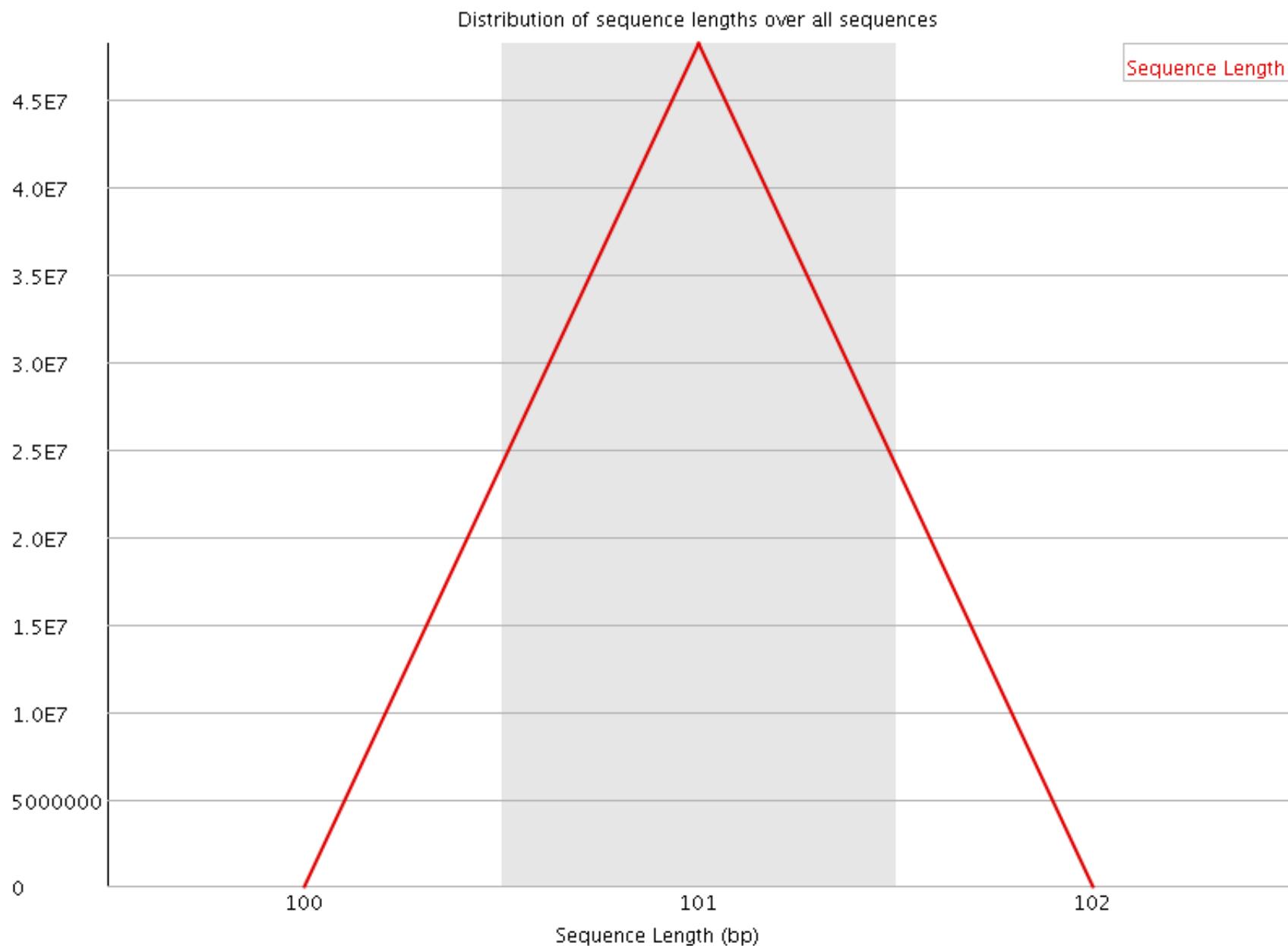
FASTQC - per seq quality



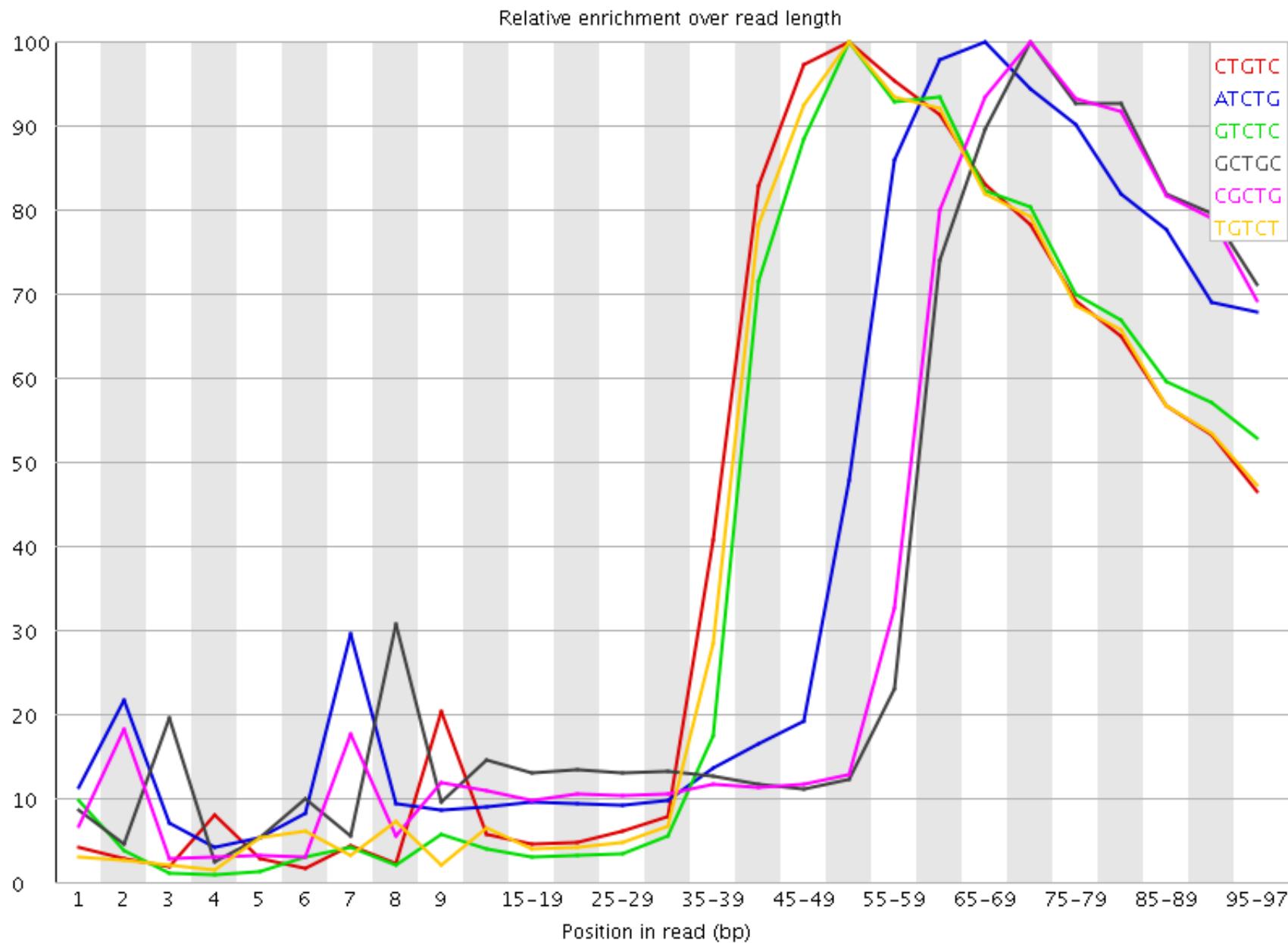
FASTQC - per seq GC content



FASTQC - Sequence Length



FASTQC - kmer distribution

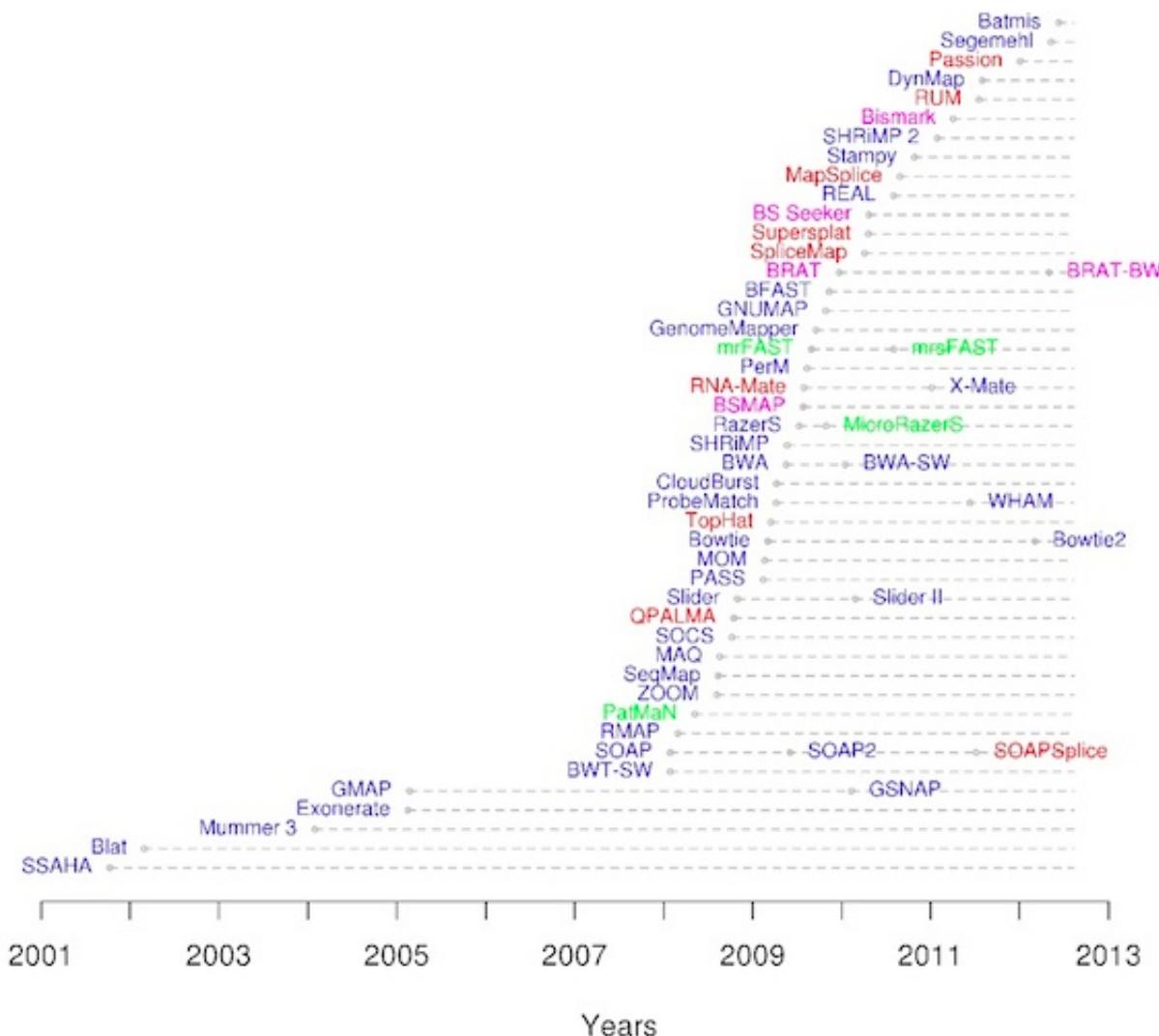


FASTQC - kmer table

Sequence	Count	Obs/Exp Overall	Obs/Exp Max	Max Obs/Exp Position
CTGTC	33437120	7.1755667	14.170156	50-54
ATCTG	33814270	7.064167	15.138542	65-69
GTCTC	32389760	6.950804	14.348899	50-54
GCTGC	29340155	6.9267426	16.531528	70-74
CGCTG	29089270	6.8675127	16.455105	70-74
TGTCT	33183170	6.6351447	13.49372	50-54
CTCTT	33408740	6.5170074	13.125135	50-54
TCTCT	33224365	6.4810414	13.289863	50-54
GCCGA	26214755	6.4660773	16.517157	75-79
GACGC	26117475	6.442083	16.140318	65-69
ATACA	30984490	6.422781	13.738384	55-59
ACATC	30017510	6.3917494	14.464595	60-64
ACACA	28701480	6.3852487	14.690713	60-64
TGCCG	27026655	6.3805614	15.88847	70-74
TACAC	29248425	6.227985	13.874619	60-64
CATCT	30438105	6.203463	13.795571	60-64
TGACG	26974620	6.1994843	15.338736	65-69
CTGAC	27494840	6.1646304	15.06331	65-69
CACAT	28919350	6.1579137	14.247532	60-64
	-----	-----	-----	-----

Getting ready to align sequence

Sequence aligners



Short read aligners

Strategy requires faster searching than BLAST or FASTA approach. Some approaches have been developed to make this fast enough for Millions of sequences. *Burrows-Wheeler Transform is a speed up that is accomplished through a transformation of the data. Requires and indexing of the search database (typically the genome). BWA, Bowtie ? LASTZ * ? BFAST*

Workflow for variant detection

- Trim
- Check quality
- Re-trim if needed
- Align
- Possibly realign around variants
- Call variants – SNPs or Indels
- Possibly calibrate or optimize with gold standard (possible in some species like Human)

NGS Alignment for DNA

- Short reads (30–200bp)
 - Bowtie and BWA – implemented with the BWT algorithm, very easy to setup and run
 - SSAHA also useful, uses fair amount of memory
 - BFAST – also good for DNA, supports Bisulfide seq,color-space but more complicated to run
- Longer reads (e.g. PacBio, 454, Sanger reads)
 - BWA has a BWA-SW mode using does a Smith-Waterman to place reads. Can tolerate large indels much better than standard BWA algorithm but slower.
 - LAST for long reads

BWA alignment choices

From BWA manual

On 350–1000bp reads, BWA-SW is several to tens of times faster than the existing programs. Its accuracy is comparable to SSAHA2, more accurate than BLAT. Like BLAT, BWA-SW also finds chimera which may pose a challenge to SSAHA2. On 10–100kbp queries where chimera detection is important, BWA-SW is over 10X faster than BLAT while being more sensitive.

BWA-SW can also be used to align ~100bp reads, but it is slower than the short-read algorithm. Its sensitivity and accuracy is lower than SSAHA2 especially when the sequencing error rate is above 2%. This is the trade-off of the 30X speed up in comparison to SSAHA2's -454 mode.

When running BWA you will also need to choose an appropriate indexing method – read the manual. This applies when your genome is very large with long chromosomes.

Colorspace alignment

- For SOLiD data, need to either convert sequences into FASTQ or run with colorspace aware aligner
 - BWA, SHRiMP, BFAST can do color-space alignment

Realignment for variant identification

- Typical aligners are optimized for speed, find best place for the read.
- For calling SNP and Indel positions, important to have optimal alignment
- Realignment around variable positions to insure best placement of read alignment
 - Stampy applies this with fast BWA alignment followed by full Smith–Waterman alignment around the variable position
 - Picard + GATK employs a realignment approach which is only run for reads which span a variable position. Increases accuracy reducing False positive SNPs.

Alignment data format

- SAM format and its Binary Brother, BAM
- Good to keep it sorted by chromosome position or by read name
- BAM format can be indexed allowing for fast random access
 - e.g. give me the number of reads that overlap bases 3311 to 8006 on chr2

Manipulating SAM/BAM

- SAMtools
 - One of the first tools written. C code with Perl bindings Bio::DB::Sam (Lincoln Stein FTW!) with simple Perl and OO-BioPerl interface
 - Convert SAM <-> BAM
 - Generate Variant information, statistics about number of reads mapping
 - Index BAM files and retrieve alignment slices of chromosome regions
- Picard – java library for manipulation of SAM/BAM files
- BEDTools – C tools for interval query in BED,GFF and many other format fiels
 - Can generate per-base or per-window coverage from BAM files with GenomeGraph
- BAMTools C++ tools for BAM manipulation and statistics

Using BWA,SAMtools

```
# index genome before we can align (only need to do this once)
$ bwa index Saccharomyces
# -t # of threads
# -q quality trimming
# -f output file
# for each set of FASTQ files you want to process these are steps
$ bwa aln -q 20 -t 16 -f SRR567756_1.sai Saccharomyces SRR567756_1.fastq
$ bwa aln -q 20 -t 16 -f SRR567756_2.sai Saccharomyces SRR567756_2.fastq
# do Paired-End alignment and create SAM file
$ bwa sampe -f SRR567756.sam Saccharomyces SRR567756_1.sai SRR567756_2.sai SRR567756_1.fastq SRR56775

# generate BAM file with samtools
$ samtools view -b -S SRR567756.sam > SRR567756.unsrt.bam
# will create SRR567756.bam which is sorted (by chrom position)
$ samtools sort SRR567756.unsrt.bam SRR567756
# build index
$ samtools index SRR567756.bam
```

BAM using Picard tools

Can convert and sort all in one go with Picard

```
$ java -jar SortSam.jar IN=SRR567756.sam OUT=SRR567756.bam SORT_ORDER=coordinate
```

Lots of other resources for SAM/BAM manipulation in Picard documentation on the web
<http://picard.sourceforge.net/command-line-overview.shtml>.

View header from BAM file

```
$ samtools view -h SRR527547.realign.W303.bam
samtools view -h SRR527547.realign.W303.bam | more
@HD VN:1.0  GO:none  SO:coordinate
@SQ SN:chrI LN:230218  UR:file:/bigdata/jstajich/Teaching/CSHL_2012_NGS/examples/genome/Saccharomyce
@SQ SN:chrII   LN:813184  UR:file:/bigdata/jstajich/Teaching/CSHL_2012_NGS/examples/genome/Saccharo

$ samtools view -bS SRR527547.sam > SRR527547.unsrt.bam
$ samtools sort SRR527547.unsrt.bam SRR527547
# this will produce SRR527547.bam
$ samtools index SRR527547.bam
$ samtools view -h @SQ  SN:chrV LN:576874
@SQ SN:chrVI   LN:270161
@SQ SN:chrVII  LN:1090940
@SQ SN:chrVIII LN:562643
@SQ SN:chrIX   LN:439888
@SQ SN:chrX    LN:745751
@SQ SN:chrXI   LN:666816
@SQ SN:chrXII  LN:1078177
@SQ SN:chrXIII LN:924431
@SQ SN:chrXIV  LN:784333
@SQ SN:chrXV   LN:1091291
@SQ SN:chrXVI  LN:948066
@SQ SN:chrMito LN:85779
@PG ID:bwa  PN:bwa  VN:0.6.2-r131
SRR527547.1387762  163 chrI    1   17  3S25M1D11M1S    =   213 260
CACCCACACCACACCCCCACACACACACACACACACACC  IIIIIIIIIHIIIIHIIIGIIIHDDG8E?@:??DDDA@
XT:A:M  NM:i:1  SM:i:17 AM:i:17 XM:i:0  X0:i:1
```

SAM format

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	[!-?A-~]{1,255}	Query template NAME
2	FLAG	Int	[0,2 ¹⁶ -1]	bitwise FLAG
3	RNAME	String	* [!-()+-<>~] [!-~]*	Reference sequence NAME
4	POS	Int	[0,2 ²⁹ -1]	1-based leftmost mapping POSition
5	MAPQ	Int	[0,2 ⁸ -1]	MAPping Quality
6	CIGAR	String	* ([0-9]+[MIDNSHPX=])+	CIGAR string
7	RNEXT	String	* = [!-()+-<>~] [!-~]*	Ref. name of the mate/next segment
8	PNEXT	Int	[0,2 ²⁹ -1]	Position of the mate/next segment
9	TLEN	Int	[-2 ²⁹ +1,2 ²⁹ -1]	observed Template LENgth
10	SEQ	String	* [A-Za-z.=.]+	segment SEQuence
11	QUAL	String	[!-~]+	ASCII of Phred-scaled base QUALity+33

Read Groups

One component of SAM files is the idea of processing multiple files, but that these track back to specific samples or replicates.

This can be coded in the header of the SAM file

```
@RG ID:Strain124 PL:Illumina PU:Genomic LB:Strain124 CN:Broad
```

It can also be encoded on a per-read basis so that multiple SAM files can be combined together into a single SAM file and that the origin of the reads can still be preserved. This is really useful when you want to call SNPs across multiple samples.

The AddOrReplaceReadGroups.jar command set in Picard is really useful for manipulating these.

samtools flagstat

```
4505078 + 0 in total (QC-passed reads + QC-failed reads)
0 + 0 duplicates
4103621 + 0 mapped (91.09%:-nan%)
4505078 + 0 paired in sequencing
2252539 + 0 read1
2252539 + 0 read2
3774290 + 0 properly paired (83.78%:-nan%)
4055725 + 0 with itself and mate mapped
47896 + 0 singletons (1.06%:-nan%)
17769 + 0 with mate mapped to a different chr
6069 + 0 with mate mapped to a different chr (mapQ>=5)
```

Realigning around Indels and SNPs

To insure high quality Indelcalls, the reads need to realigned after placed by BWA or other aligner. This can be done with PicardTools and GATK.

Need to Deduplicate reads

```
$ java -jar picard-tools/MarkDuplicates.jar INPUT=STRAIN.sorted.bam OUTPUT=STRAIN.dedup.bam  
METRICS_FILE=STRAIN.dedup.metrics CREATE_INDEX=true VALIDATION_STRINGENCY=SILENT;
```

Then identify Intervals around variants

```
$ java -jar GATK/GenomeAnalysisTK.jar -T RealignerTargetCreator -R genome/Saccharomyces_cerevisiae.fa  
-o STRAIN.intervals -I STRAIN.dedup.bam;
```

Then realign based on these intervals

```
$ java -jar GATK/GenomeAnalysisTK.jar -T IndelRealigner -R genome/Saccharomyces_cerevisiae.fa \  
-targetIntervalsSTRAIN.intervals -I STRAIN.dedup.bam -o STRAIN.realign.bam
```

SAMtools and VCFtools to call SNPs

```
$ samtools mpileup -D -S -gu -f genome/Saccharomyces_cerevisiae.fa ABC.bam | bcftools view -bvcg - > bcftools view ABC.raw.bcf | vcfutils.pl varFilter -D100 > ABC.filter.vcf
```

GATK to call SNPs

```
# run GATK with 4 threads (-nt)
# call SNPs only (-glm, would specific INDEL for Indels or can ask for BOTH)
$ java -jar GenomeAnalysisTKLite.jar -T UnifiedGenotyper -glm SNP -I SRR527545.bam \
-R genome/Saccharomyces_cerevisiae.fa -o SRR527545.GATK.vcf -nt 4
```

GATK to call INDELS

```
# run GATK with 4 threads (-nt)
# call SNPs only (-glm, would specific INDEL for Indels or can ask for BOTH)
$ java -jar GenomeAnalysisTKLite.jar -T UnifiedGenotyper -glm INDEL -I SRR527545.bam \
-R genome/Saccharomyces_cerevisiae.fa -o SRR527545.GATK_INDEL.vcf -nt 4
```

VCF Files

Variant Call Format – A standardized format for representing variations. Tab delimited but with specific ways to encode more information in each column.

```
##FORMAT=<ID=AD,Number=.,Type=Integer,Description="Allelic depths for the ref and alt alleles in the
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Approximate read depth (reads with MQ=255 or with
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=PL,Number=G,Type=Integer,Description="Normalized, Phred-scaled likelihoods for genotypes
##INFO=<ID=AC,Number=A,Type=Integer,Description="Allele count in genotypes, for each ALT allele, in t
##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency, for each ALT allele, in the same ord

#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT SRR527545
chrI 141 . C T 47.01 . AC=1;AF=0.500;AN=2;BaseQRankSum=-0.203;DP=23;Dels=0.00;
FS=5.679;HaplotypeScore=3.4127;MLEAC=1;MLEAF=0.500;MQ=53.10;MQ0=0;MQRankSum=-2.474;QD=2.04;ReadPosRan
SB=-2.201e+01 GT:AD:DP:GQ:PL 0/1:19,4:23:77:77,0,565

chrI 286 . A T 47.01 . AC=1;AF=0.500;AN=2;BaseQRankSum=-0.883;DP=35;Dels=0.00;
FS=5.750;HaplotypeScore=0.0000;MLEAC=1;MLEAF=0.500;MQ=46.14;MQ0=0;MQRankSum=-5.017;QD=1.34;ReadPosRan
SB=-6.519e-03 GT:AD:DP:GQ:PL 0/1:20,15:35:77:77,0,713
```

Filtering Variants

GATK best Practices <http://www.broadinstitute.org/gatk/guide/topic?name=best-practices> emphasizes need to filter variants after they have been called to removed biased regions.

These refer to many combinations of information. Mapping quality (MQ), Homopolymer run length (HRun), Quality Score of variant, strand bias (too many reads from only one strand), etc.

```
-T VariantFiltration -o STRAINS.filtered.vcf
--variant STRAINS.raw.vcf \
--clusterWindowSize 10 -filter "QD<8.0" -filterName QualByDepth \
-filter "MQ>=30.0" -filterName MapQual \
-filter "HRun>=4" -filterName HomopolymerRun \
-filter "QUAL<100" -filterName QScore \
-filter "MQ0>=10 && ((MQ0 / (1.0 * DP)) > 0.1)" -filterName MapQualRatio \
-filter "FS>60.0" -filterName FisherStrandBias \
-filter "HaplotypeScore > 13.0" -filterName HaplotypeScore \
-filter "MQRankSum < -12.5" -filterName MQRankSum \
-filter "ReadPosRankSum < -8.0" -filterName ReadPosRankSum >& output.filter.log
```

VCFtools

A useful tool to JUST get SNPs back out from a VCF file is vcf-to-tab (part of vcftools).

```
$ vcf-to-tab < INPUT.vcf > OUTPUT.tab
```

```
#CHROM POS REF SRR527545
chrI 141 C C/T
chrI 286 A A/T
chrI 305 C C/G
chrI 384 C C/T
chrI 396 C C/G
chrI 476 G G/T
chrI 485 T T/C
chrI 509 G G/A
chrI 537 T T/C
chrI 610 G G/A
chrI 627 C C/T
```

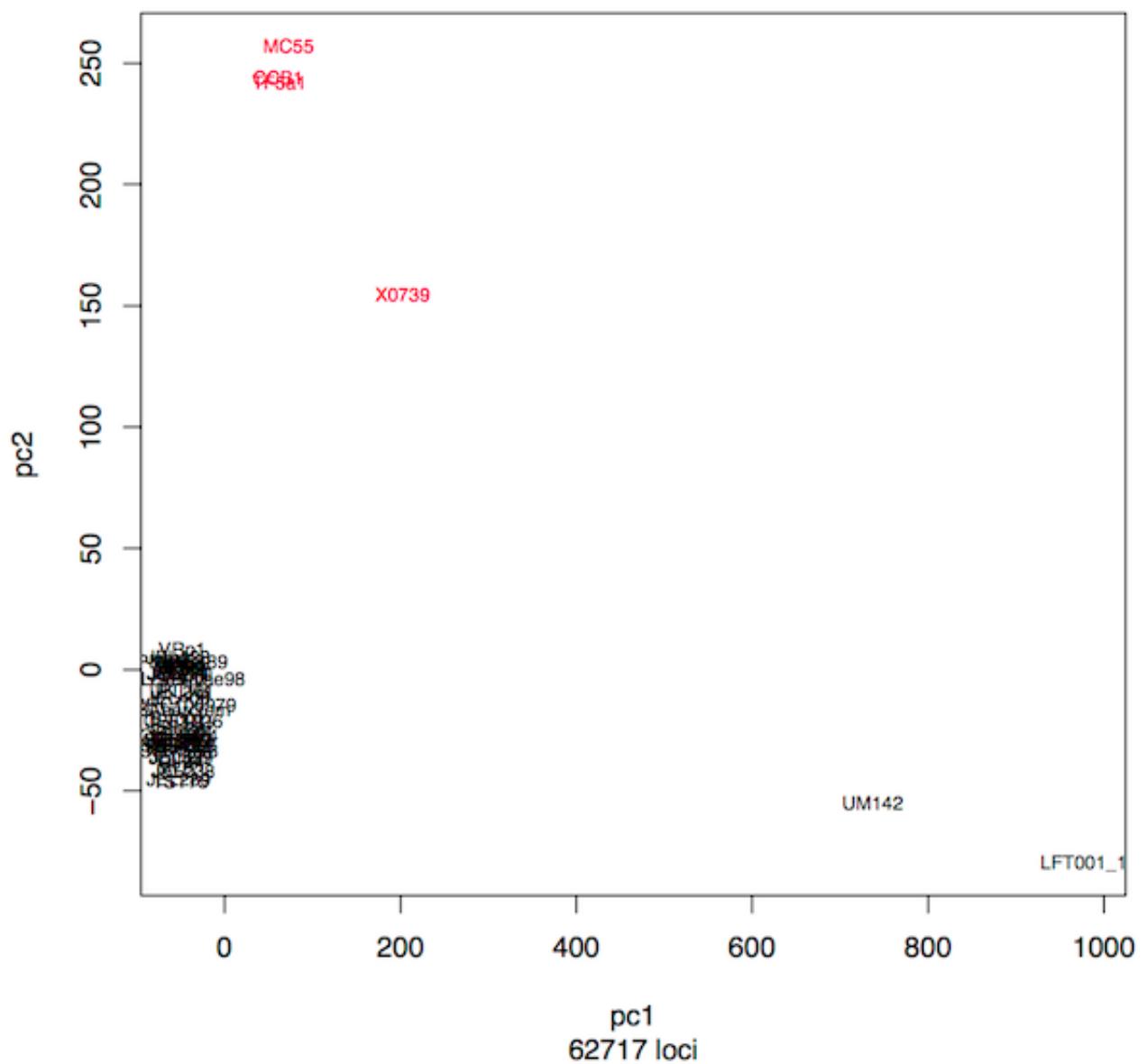
VCFtools to evaluate and manipulate

```
$ vcftools --vcf SRR527545.GATK.vcf --diff SRR527545.filter.vcf
N_combined_individuals: 1
N_individuals_common_to_both_files: 1
N_individuals_unique_to_file1: 0
N_individuals_unique_to_file2: 0
Comparing sites in VCF files...
Non-matching REF at chrI:126880 C/CTTTTTTTTTTTTT. Diff results may be unreliable.
Non-matching REF at chrI:206129 A/AAC. Diff results may be unreliable.
Non-matching REF at chrIV:164943 C/CTTTTTTTTTTT. Diff results may be unreliable.
Non-matching REF at chrIV:390546 A/ATTGTTGTTGT. Diff results may be unreliable.
Non-matching REF at chrXII:196750 A/ATTTTTTTTTTTTT. Diff results may be unreliable.
Found 8604 SNPs common to both files.
Found 1281 SNPs only in main file.
Found 968 SNPs only in second file.

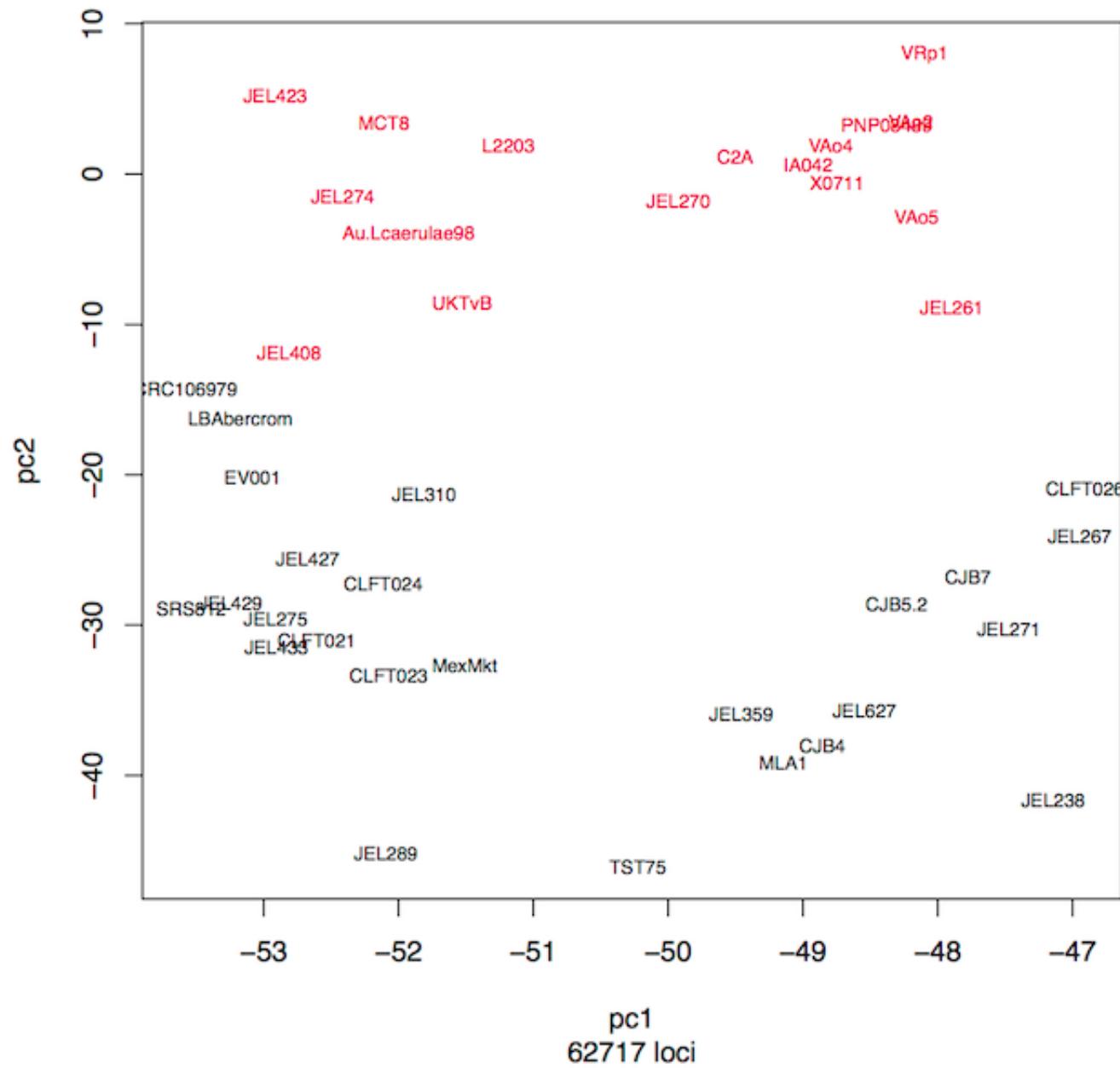
# calculate Tajima's D in binsizes of 1000 bp [if you have multiple individuals]
$ vcftools --vcf Sacch_strains.vcf --TajimaD 1000
```

Can compare strains in other ways

PCA plot of strains from the SNPs converted to 0,1,2 for homozygous Ref, Homozygous Alt allele, or heterozygous (done in R)

PCA: 48 Isolates SNPs – filtered data – Loci w/ Missing Values Excluded

Zoomed PCA plot



Summary

- Reads should be trimmed, quality controlled before use. Preserving Paired-End info is important
- Alignment of reads with several tools possible, BWA outlined here
- SAMTools and Picard to manipulate SAM/BAM files
- Genotyping with SAMtools and GATK
- Summarizing and manipulating VCF files with VCFtools

Structural variation

Programming for Biology

CSH, October 2012

Tomas Marques-Bonet
ICREA Research Professor
Institut de Biologia Evolutiva

Continuum of Genomic Variation

Nucleotide

- Single base-pair changes
 - Point mutations (1 per 800 bp)
- Small insertions/deletions
 - Frameshift, microsatellite, minisatellite
- Mobile elements
 - Retroelement insertions (300bp -10 kb)
- Large-scale genomic copy number variation (>10 kb)
 - Large-scale Deletions
 - Segmental Duplications
- Local Rearrangements
- Chromosomal variation
 - Translocation, inversion, fusion

Cytogenetics

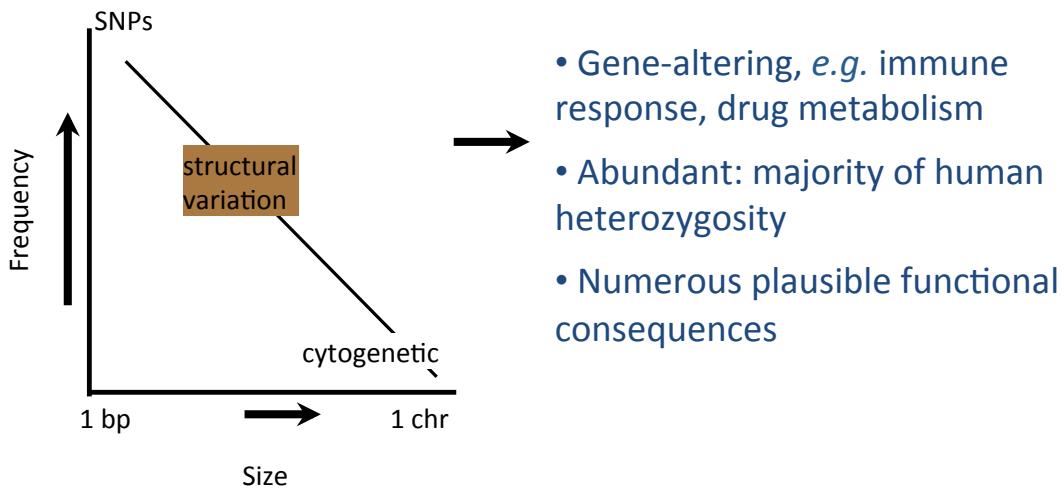


Copy Number Variation

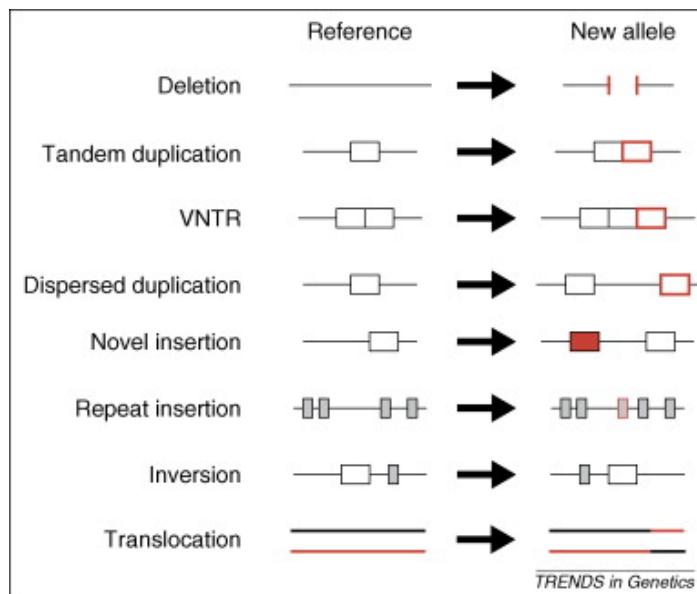
Structural Variants (SV)

Genomic Structural Variation

Human Genetic Variation

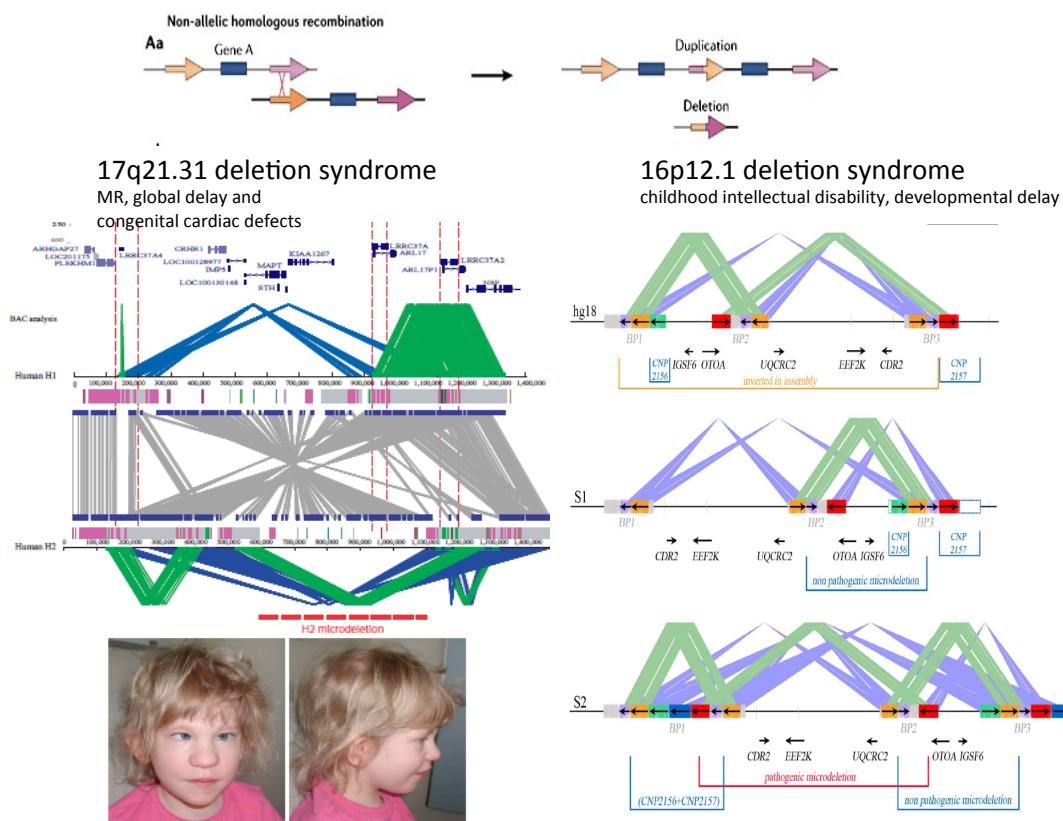


Types of Structural Variation



Why Study Structural Variation?

- Common in “normal” human genomes-- major cause of phenotypic variation
- Common in certain diseases, particularly cancer
- Now showing up in rare disease; autism, schizophrenia



Zody et al. Nature Genetics (2008)

Antonacci et al. Nature Genetics (2010)

Challenges of CNV studies

- Often involves repeated regions
- Rearrangements are complex
- Can involve highly repetitive elements

Methods to Find SVs

Experimental approach

ArrayCGH (SNP based and genomic)

Sequence based

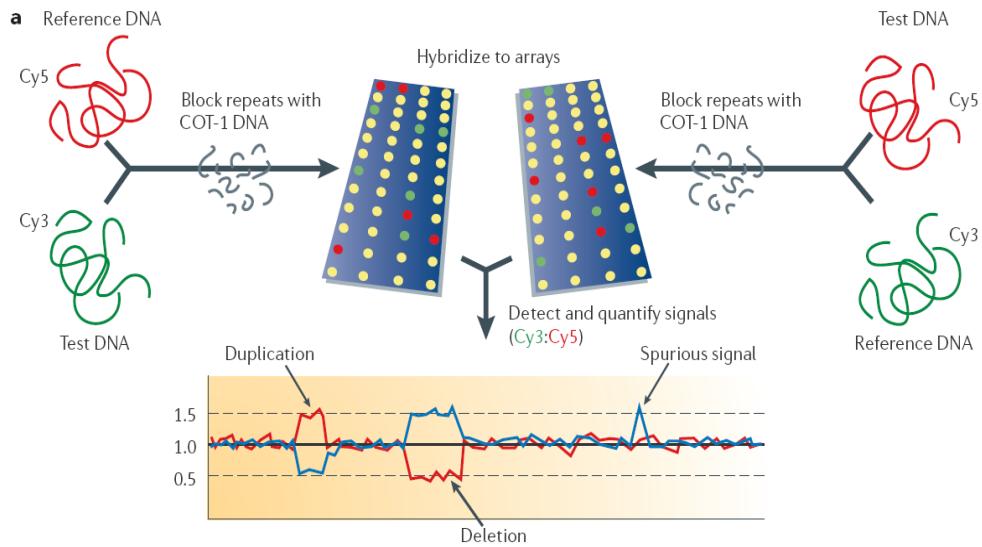
Local and *de novo* assembly

Read pair analysis

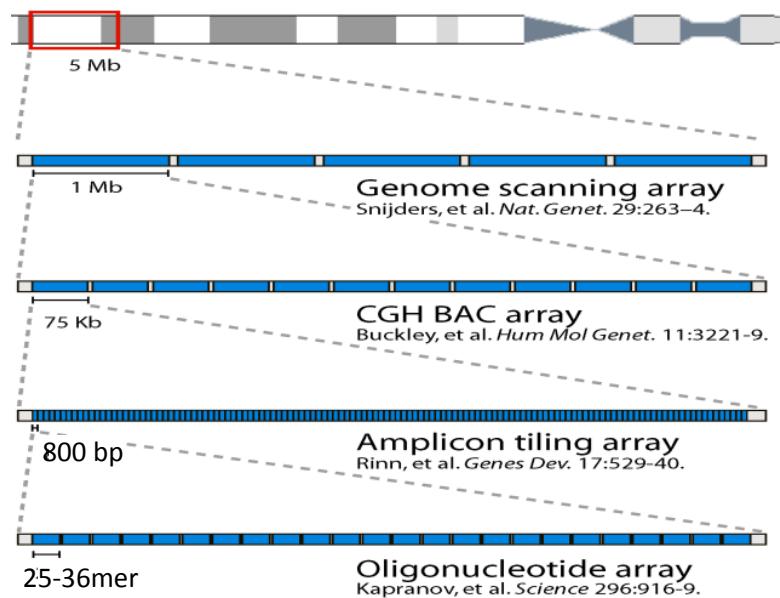
Read depth analysis

Split read analysis

METHOD 1: Copy Number Variation: Array Comparative Genomic Hybridization



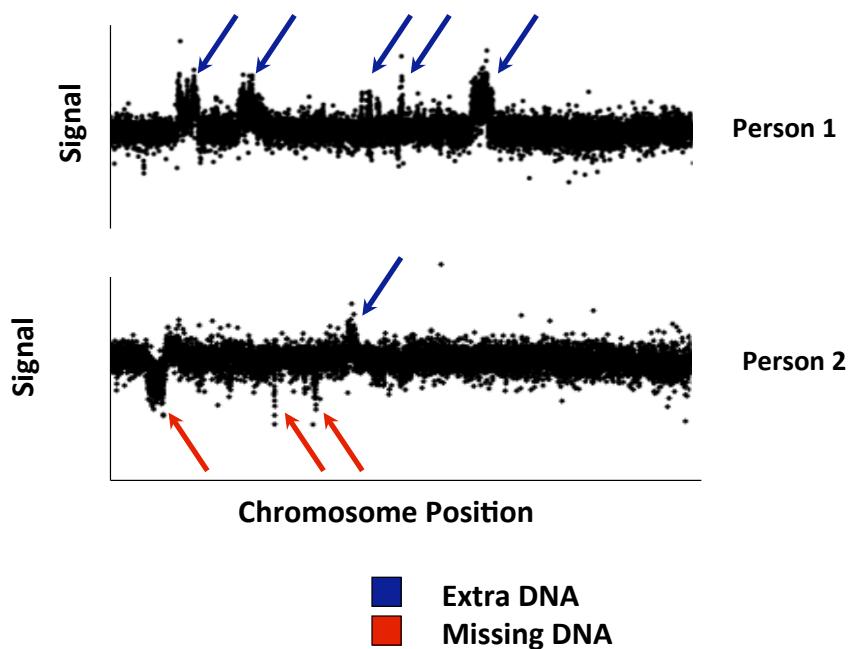
Genome Tiling Arrays



Typical Analysis Procedure

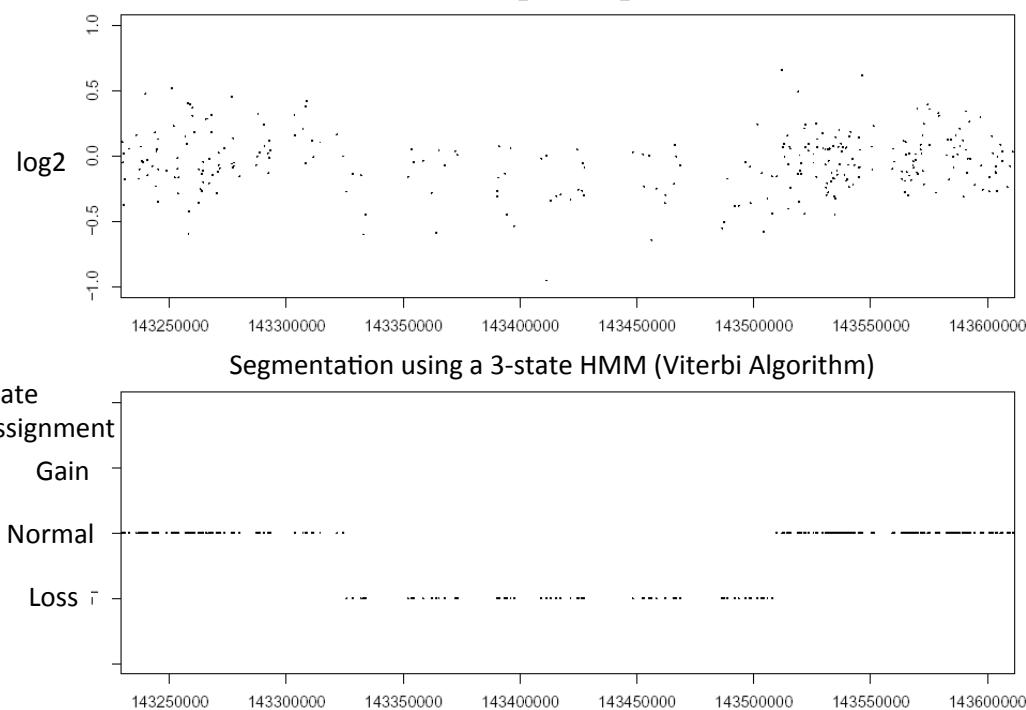
- For each probe, calculate a log2 ratio of test/reference
 - Log2 serves to center values around 0
 - Hemizygous deletion in test: $\log_2(\text{test}/\text{reference}) = \log_2(1/2) = -1$
 - Duplication in test:
 $\log_2(\text{test}/\text{reference}) = \log_2(3/2) = 0.59$
 - Homozygous duplication:
 $\log_2(\text{test}/\text{reference}) = \log_2(4/2) = 1$

Copy Number Variations in the Human Genome

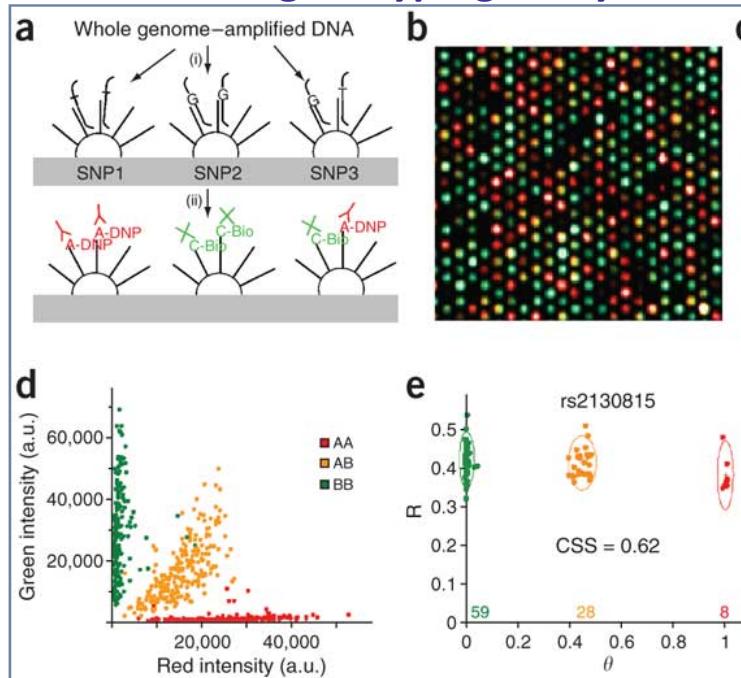


3-State HMM

chr07.64797_143243594_143597470

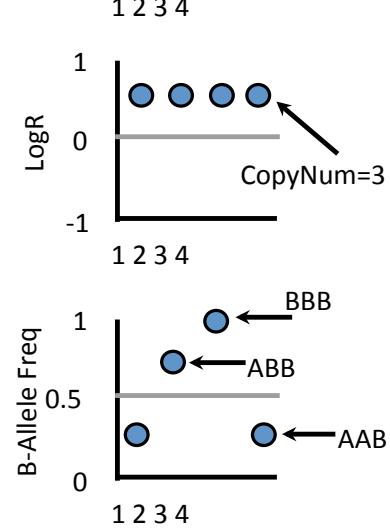
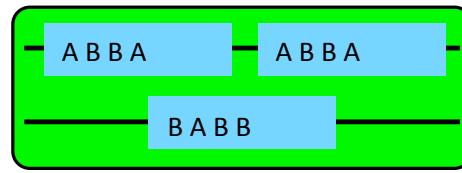
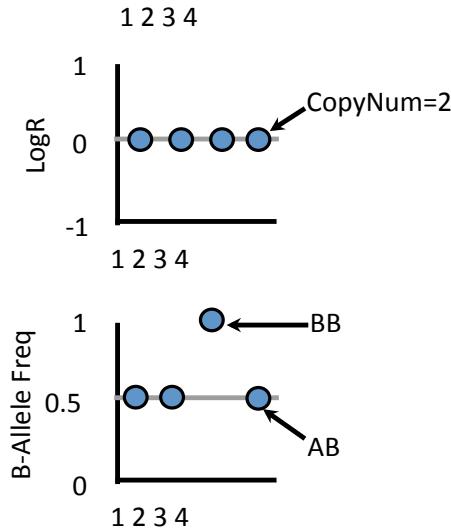
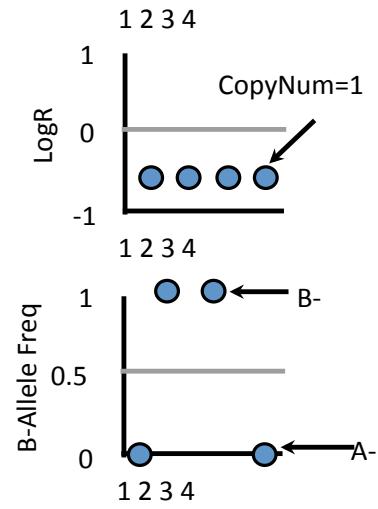
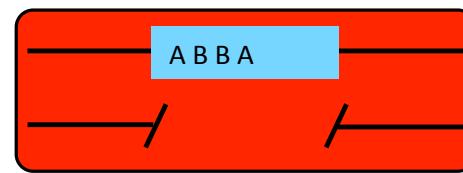
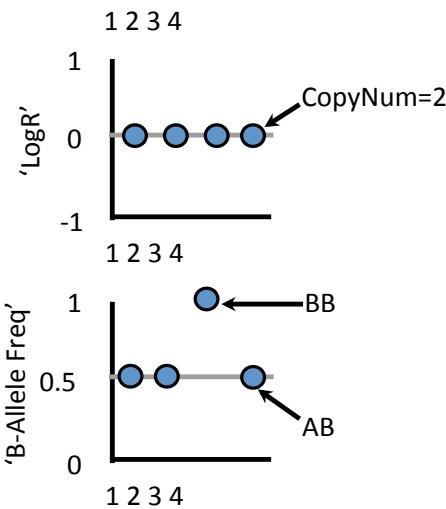
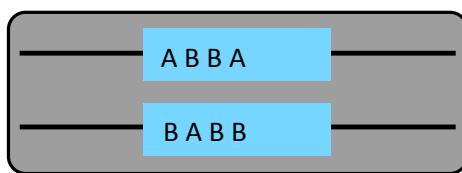


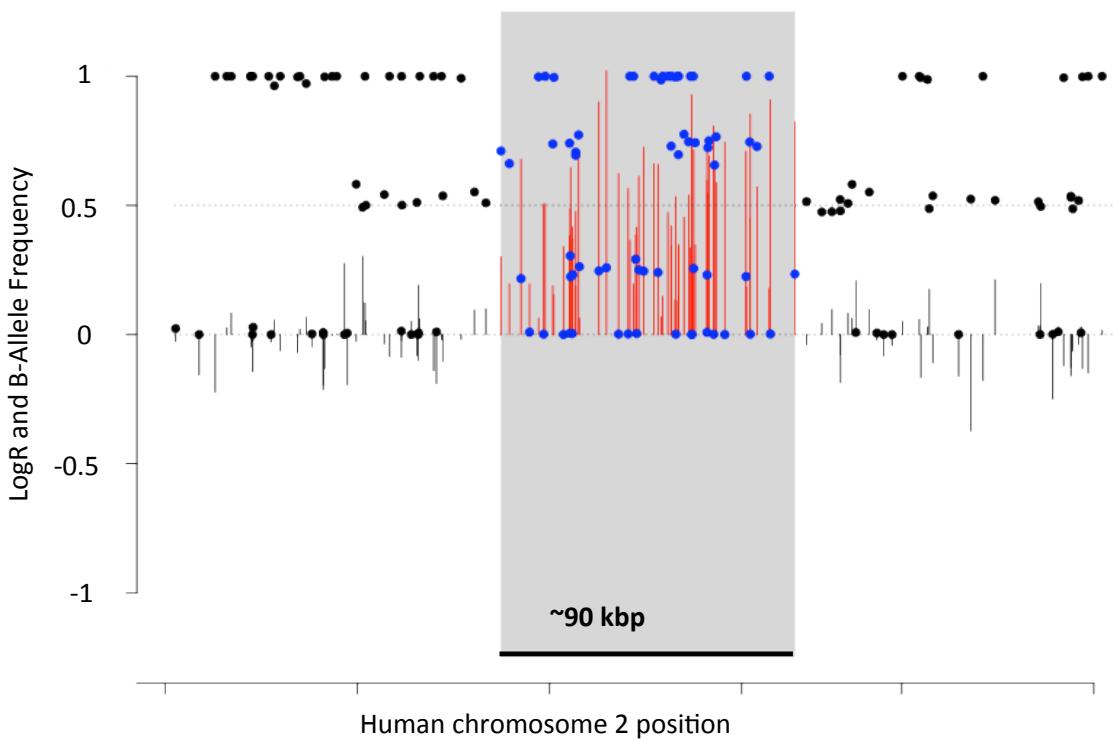
METHOD 2: Copy Number Variation: SNP genotyping Array



Steemers et al.

SNP Fluorescence-Based Deletion Discovery





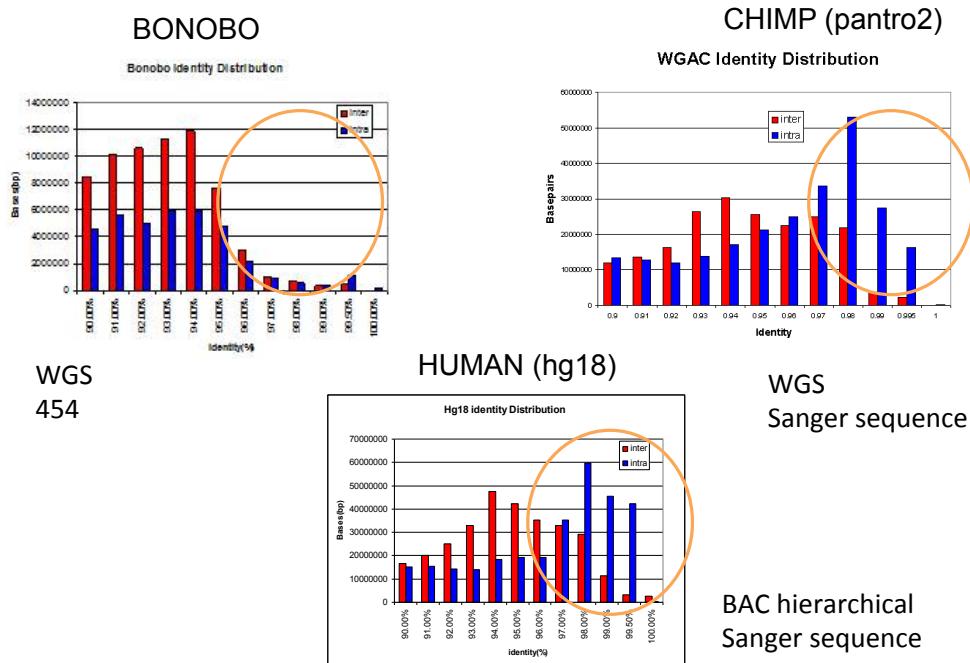
Sequencing Methods

- Going Backwards... Sanger, 454, Illumina.....
- CNV and SV are hotspots of research... but reality is:
 - Limitations of the methods
 - Indirect methods. ALL have problems!!
 - What do we want?
 - Clone approached
 - Finish sequence

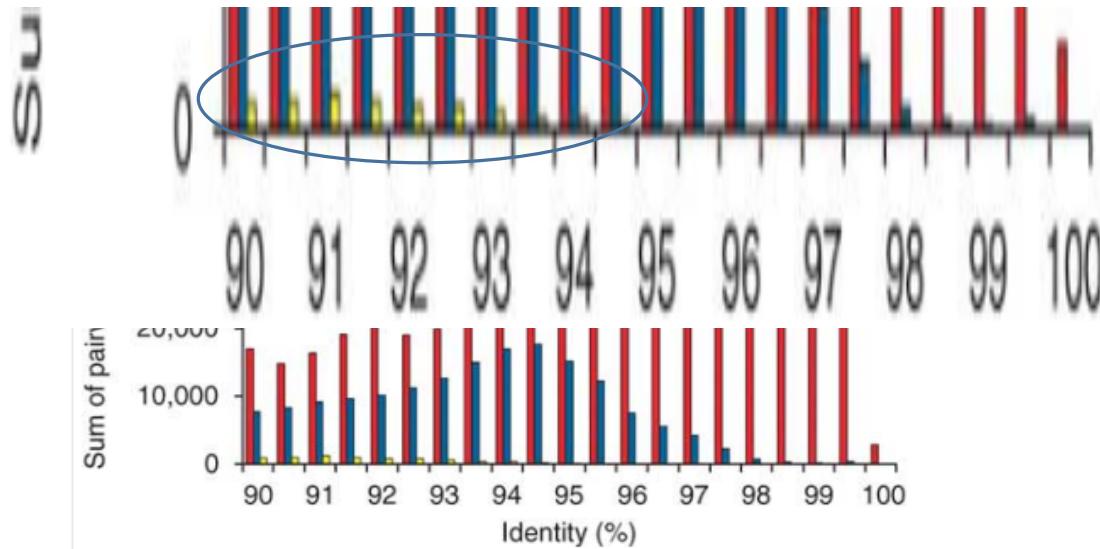
De novo assemblies

- Theory vs. Reality
- Most assemblies (even with Sanger technology!) are collapsed.
- Examples....

Distribution of Total duplications

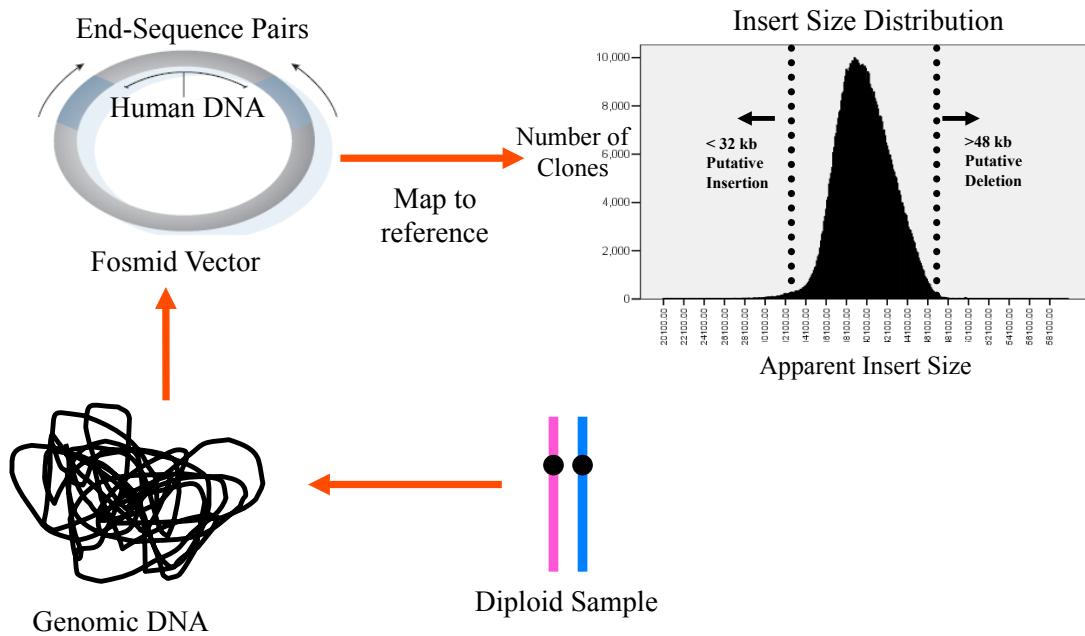


Limitations of NGS assemblies



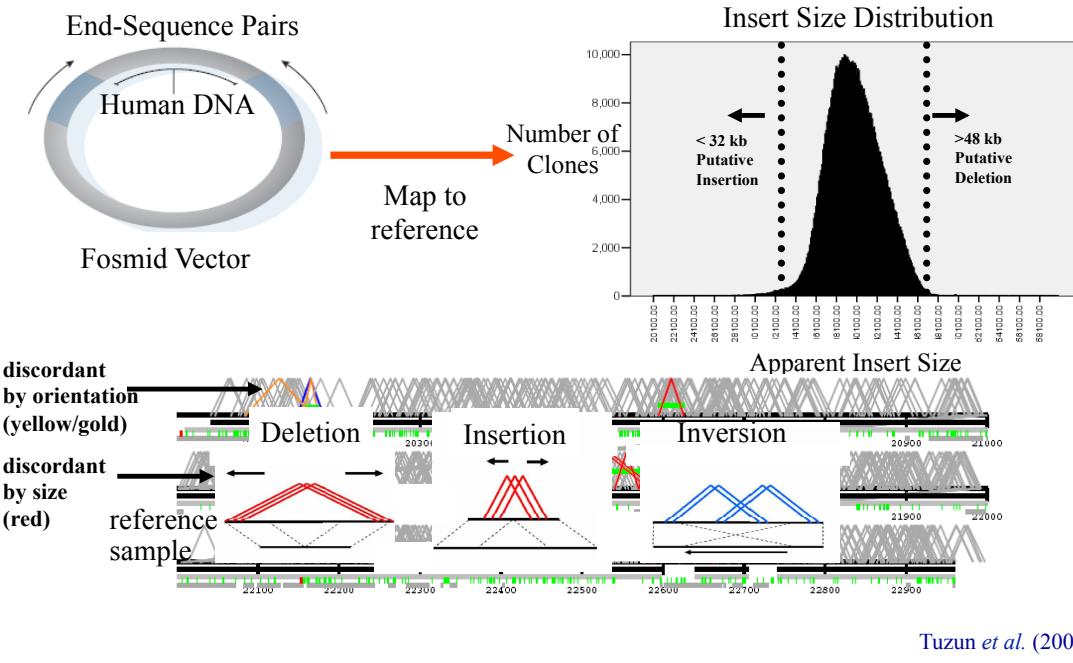
Alkan et al. Nature Methods 2010

Method 2: End-Sequence Pair (ESP) Analysis



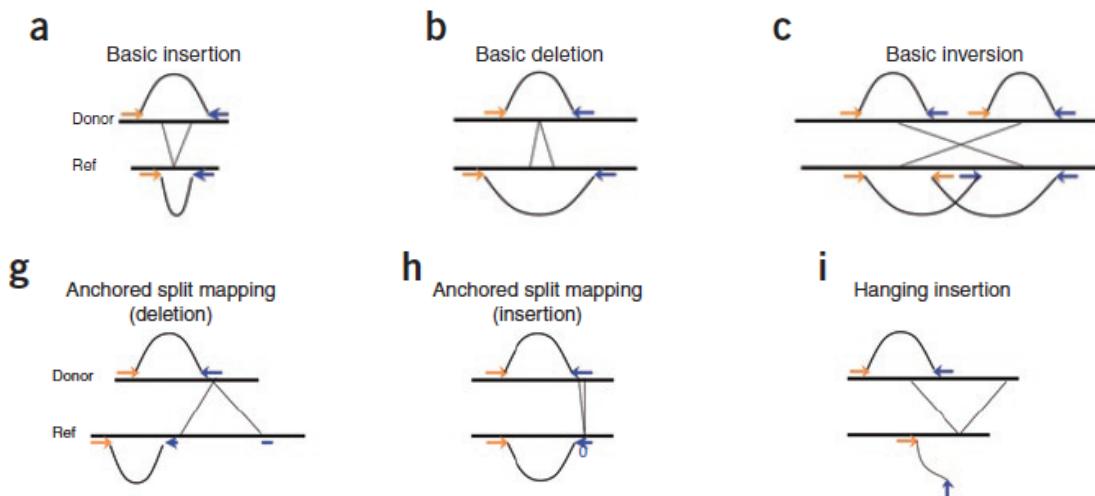
Tuzun et al. (2005)

Method 2: End-Sequence Pair (ESP) Analysis

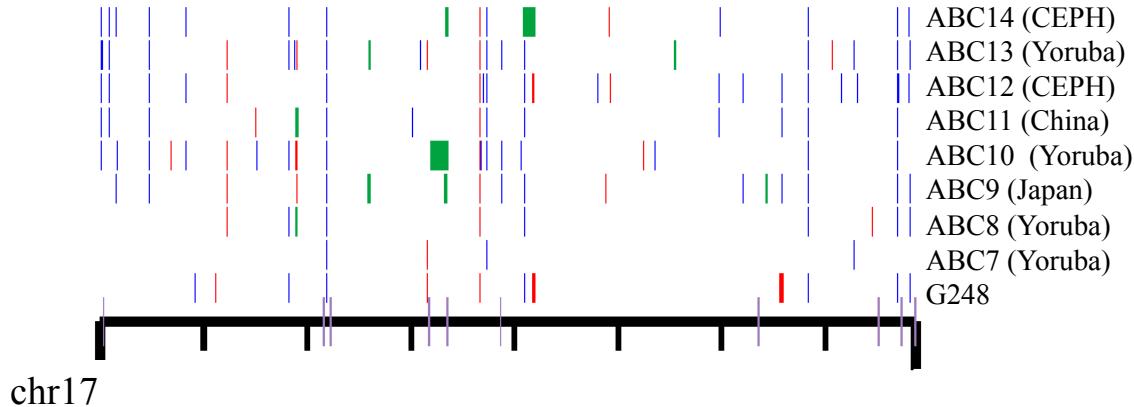


What can we find?

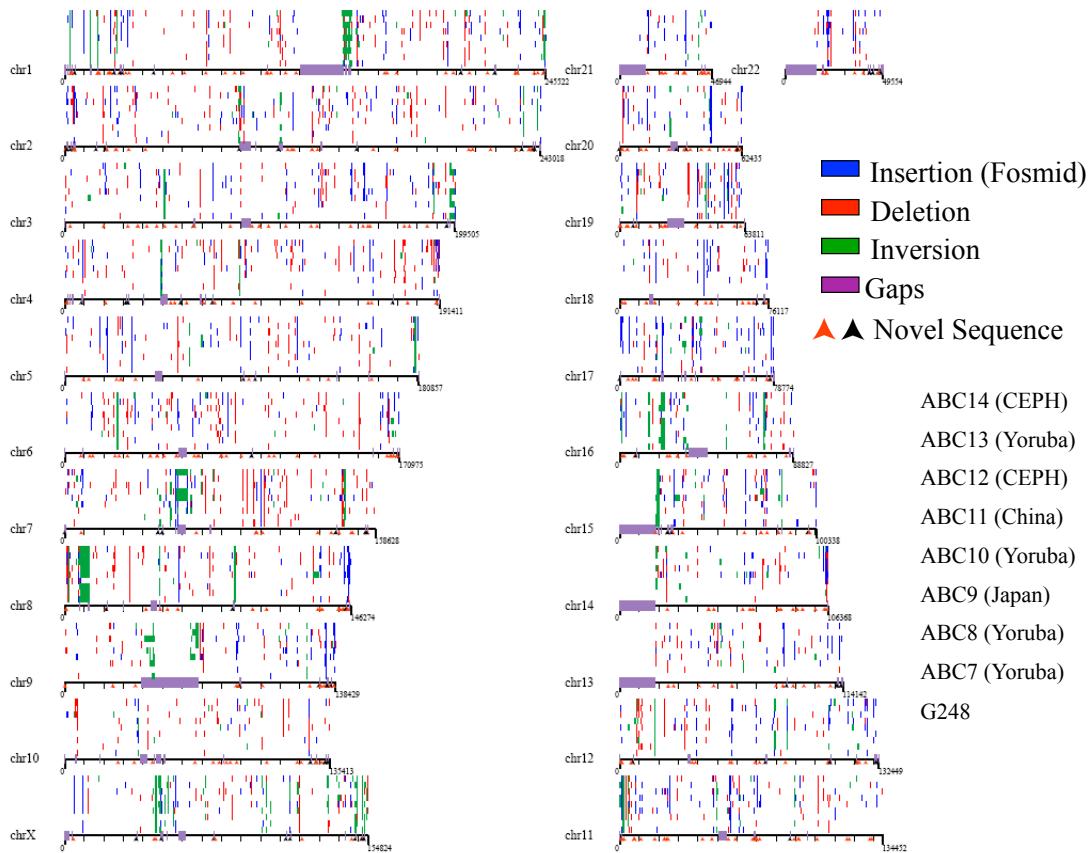
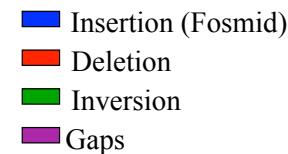
Structural variation detection:



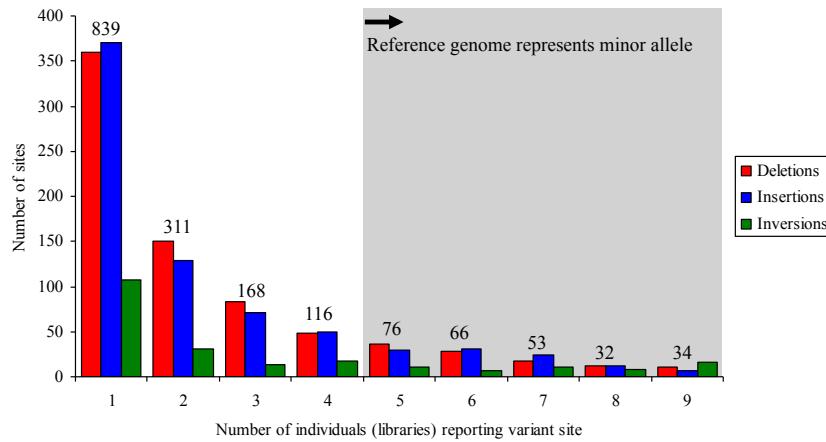
Map of Validated Variants



- Genome wide map of variants
- Ability to resolve structure of individual haplotypes

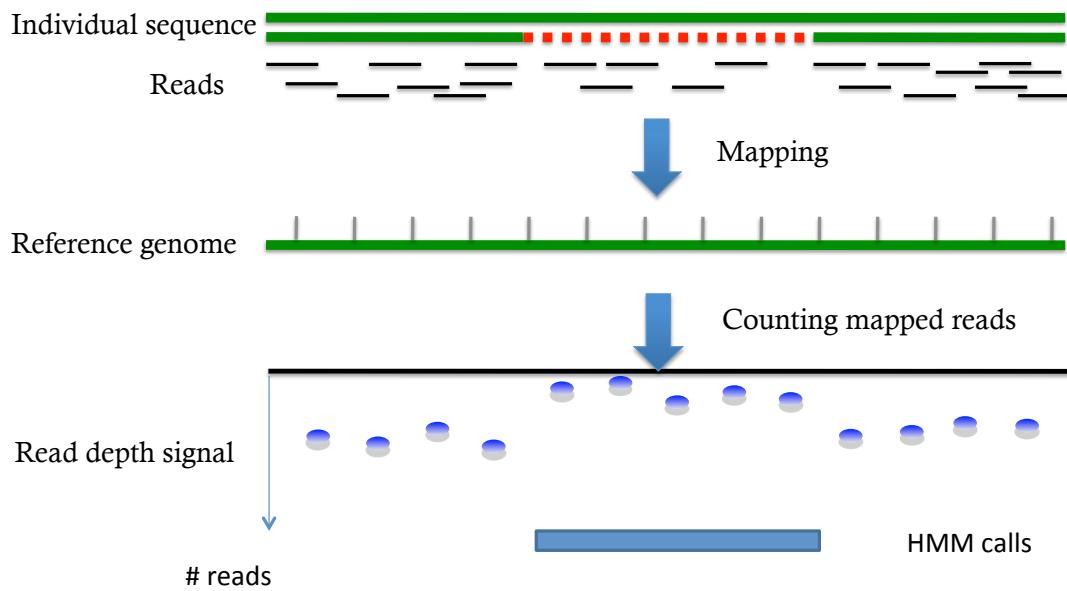


Frequency of Validated Sites

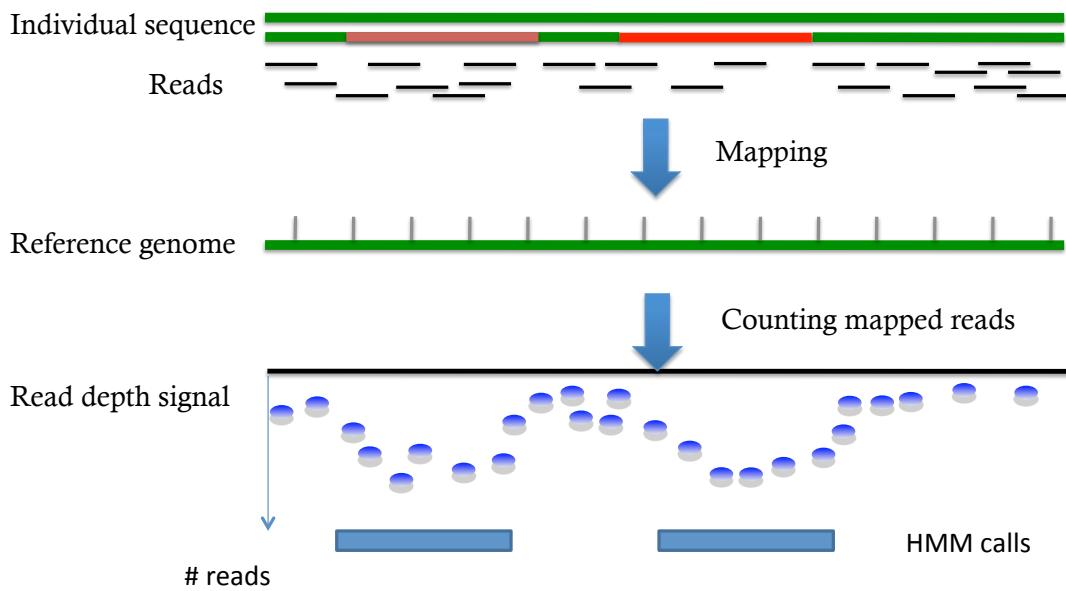


261 (15%) sites where reference genome represents a minor allele

Method 3: Sequence Read Depth Analysis

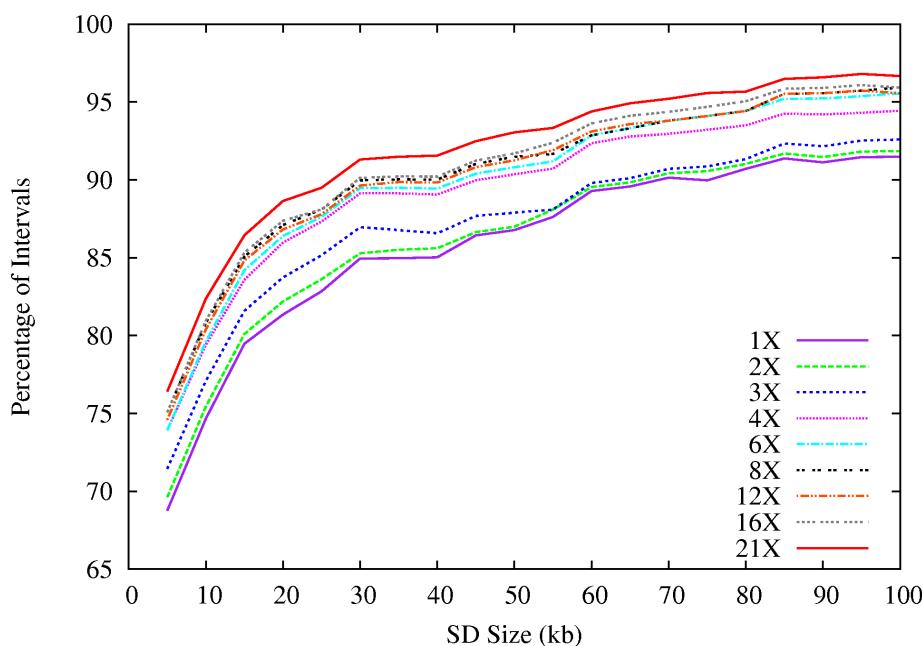


Method 3: Sequence Read Depth Analysis

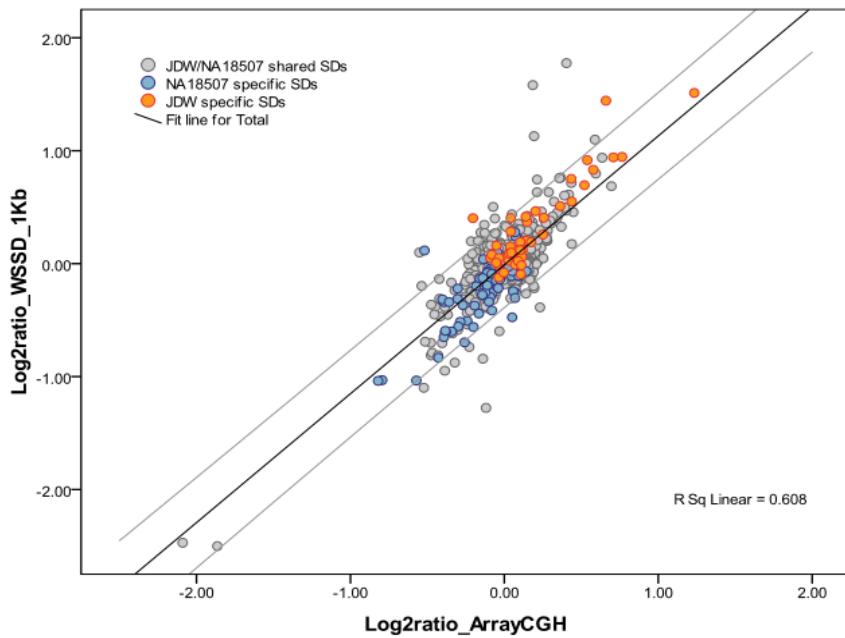


29

Sequence coverage and detection power

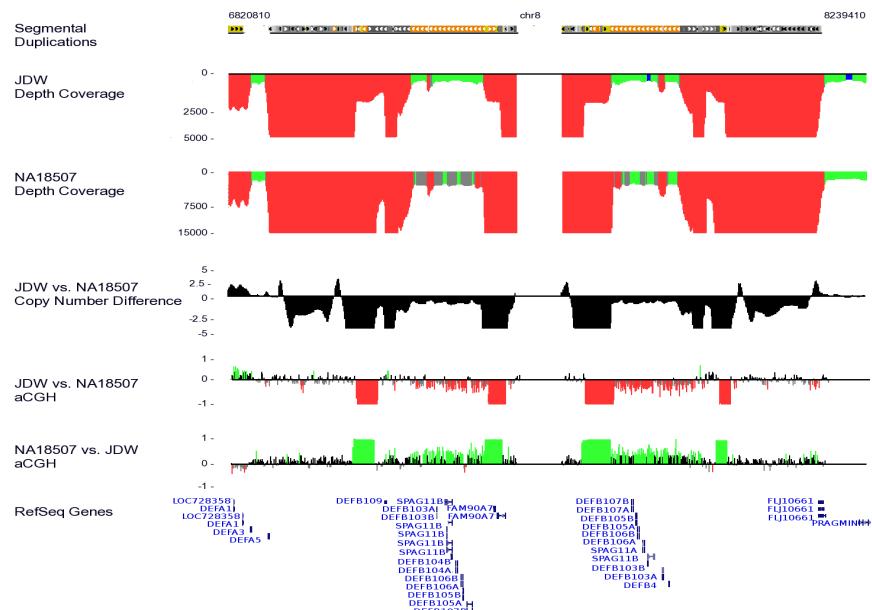


Validation of copy-number estimations



Alkan et al., *Nature Genetics*, 2009

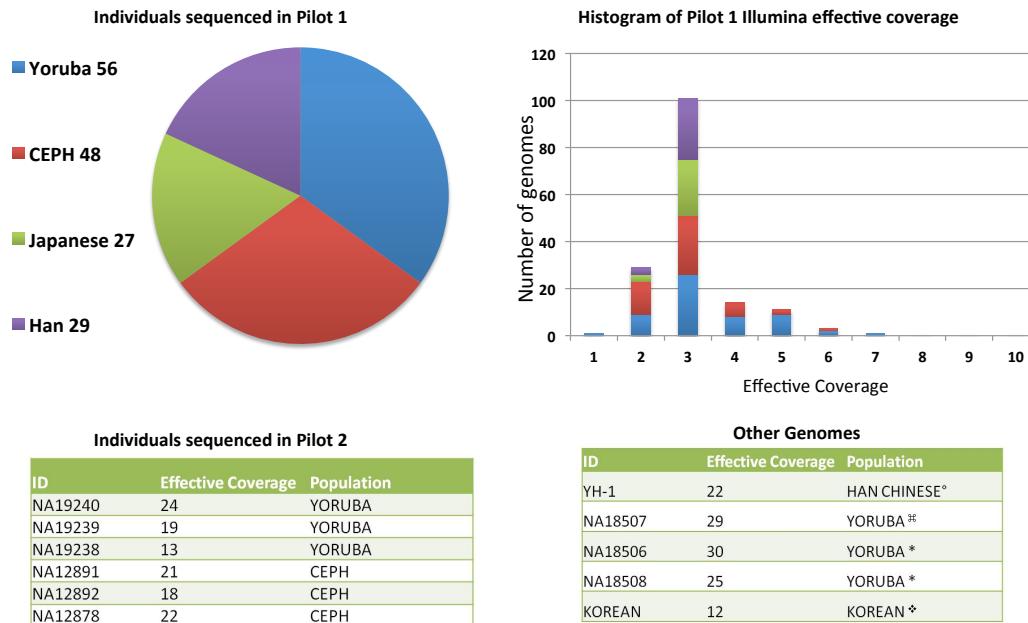
Defensin gene cluster + *FAM90A7*



Associated with psoriasis and Crohn's disease

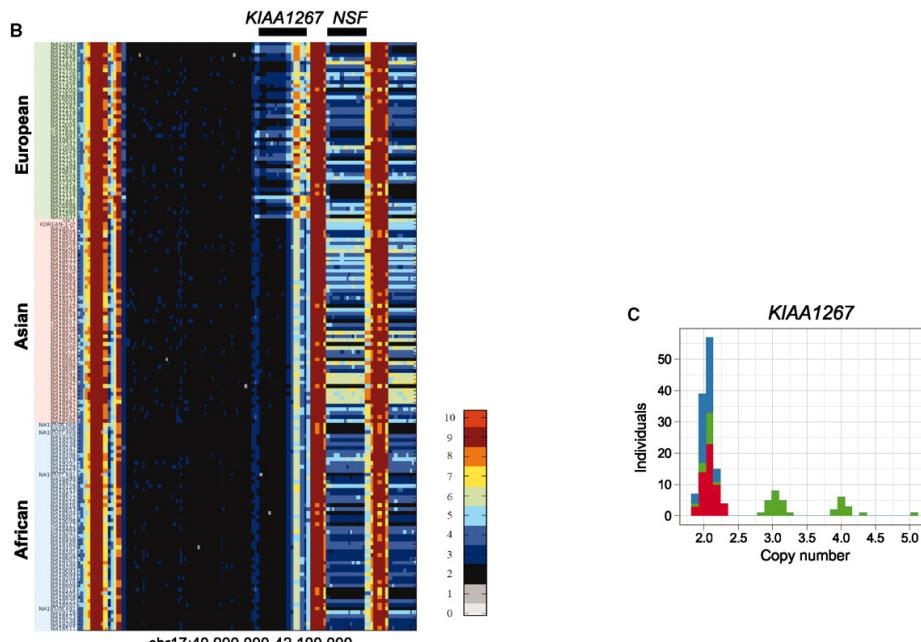
Alkan et al. Nature Genetics 2009

Scaling up: 1000 Genomes and more



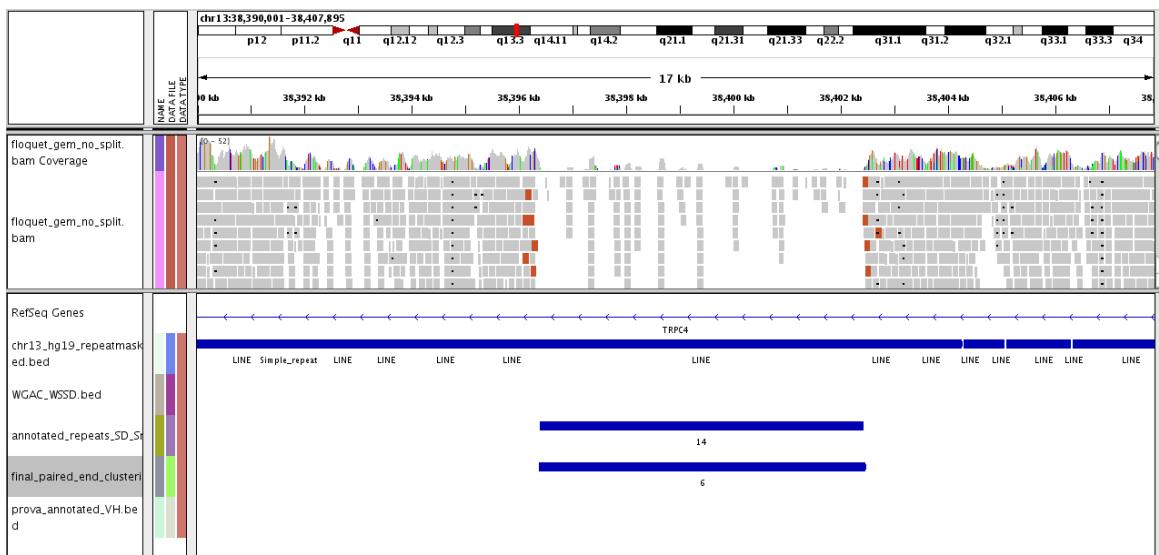
Sudmant, Kitzman, et al., *Science*, 2010

Copy number variation in human populations

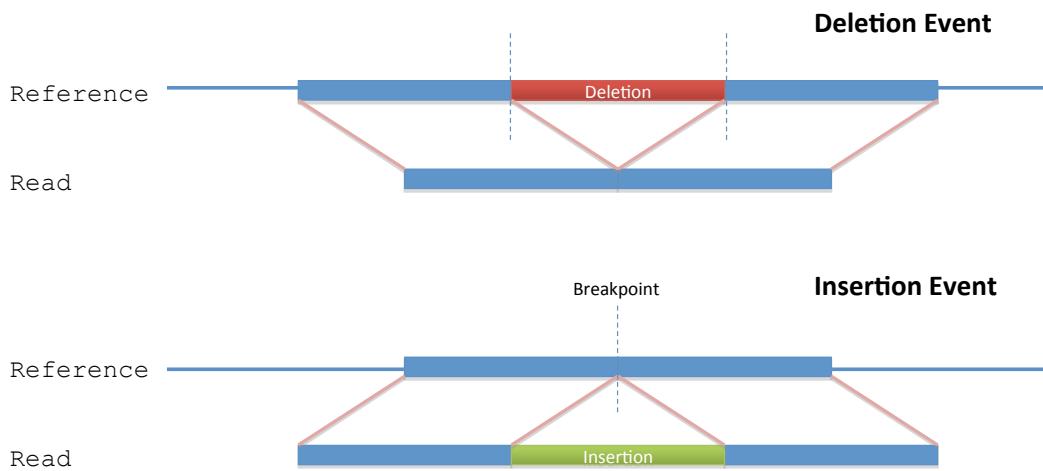


Sudmant et al. *Science* 2010

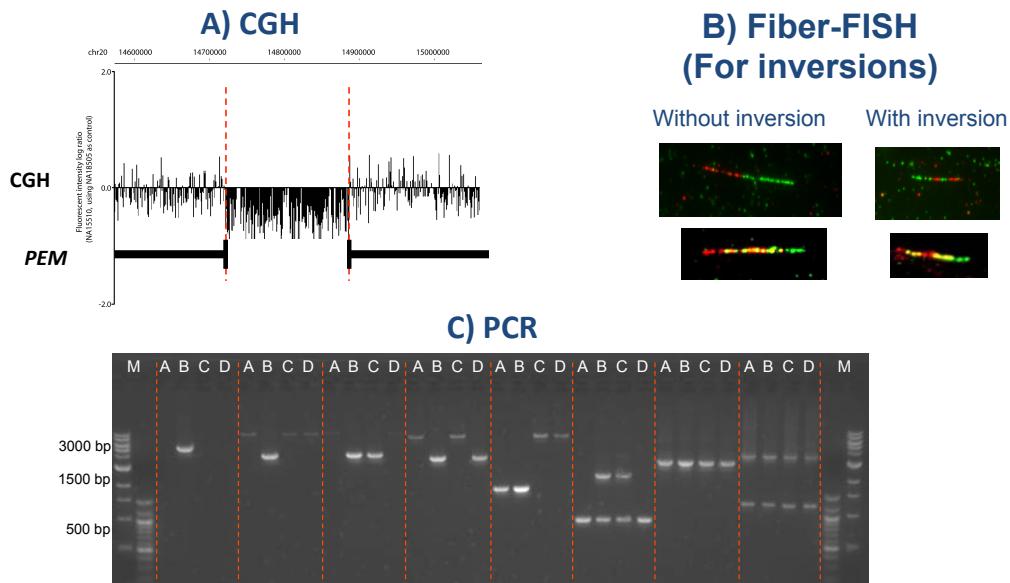
Deletions



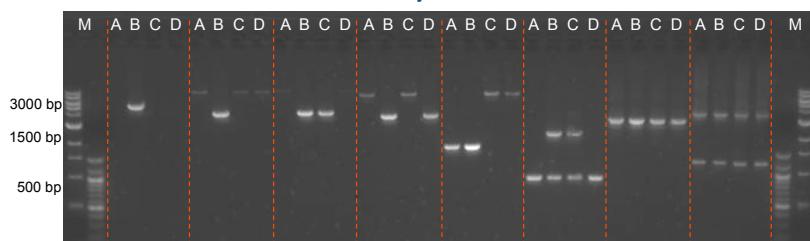
Split-read Analysis



Experimental Validation



C) PCR



Methods to Find SVs

Experimental approach

ArrayCGH (SNP based and genomic)

Based on ratios, Saturate quite fast, poor breakpoint resolution

Sequence based

Read pair analysis

Deletions, small novel insertions, inversions, transposons

Size and breakpoint resolution dependent to insert size

Read depth analysis

Deletions and duplications

Relatively poor breakpoint resolution

Split read analysis

Small novel insertions/deletions, and mobile element insertions

1bp breakpoint resolution

Local and *de novo* assembly

SV in unique segments

1bp breakpoint resolution

Programming for Biology Protein Evolution / Similarity Searching

What BLAST Does / Why BLAST works

Bill Pearson
wrp@virginia.edu

1

Sequence Similarity - Conclusions

- Homologous sequences share a common ancestor, but most sequences are non-homologous
- Always compare Protein Sequences
- Sequence Homology can be reliably inferred from statistically significant similarity (non-homology cannot from non-similarity)
- Homologous proteins share common structures, but not necessarily common functions
- Sequence statistical significance estimates are accurate (verify this yourself) $10^{-6} < E() < 10^{-3}$ is statistically significant
- Scoring matrices set evolutionary look back horizons - not every discovery is distant
- PSI-BLAST can be more sensitive, but with lower statistical accuracy

2

1

Establishing homology from statistically significant similarity

Why BLAST works

- For most proteins, homologs are easily found over long evolutionary distances (500 My – 2 By) using standard approaches (BLAST, FASTA)
- Difficult for distant relationships or very short domains
- Most default search parameters are optimized for distant relationships and work well

3

This talk is not about:

- *Alignment*
 - Alignment quality may be more sensitive to parameter choice
 - Multiple sequences for biologically accurate alignments
- *Inferring Protein Function*
 - Homology (common ancestry) implies common structure (guaranteed), not necessarily common function
 - Homologs have different functions
 - Non-homologs have similar (or identical) functions
- *The best sequences for building evolutionary trees*
 - Protein sequences are clearly best for establishing homology, but DNA sequences may be better for resolving recent divergence

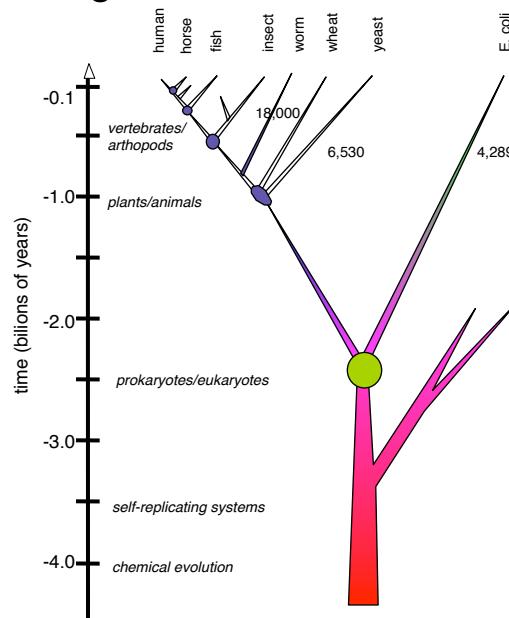
4

Protein Evolution and Sequence Similarity

- What is Homology and how do we recognize it?
- How do we measure sequence similarity – alignments and scoring matrices?
- DNA vs protein comparison
- Alignment Algorithms/Local sequence alignments
- Similarity scoring matrices
- When are we certain that an alignment is significant - similarity score statistics?
- When to trust similarity statistics?
- Improving sensitivity with PSI-BLAST

5

Homologues share a common ancestor



6

When do we infer homology?

Homology \Leftrightarrow structural similarity
? sequence similarity

Bovine trypsin (5ptp)
 Structure: $E() < 10^{-23}$,
 RMSD 0.0 Å
 Sequence: $E() < 10^{-84}$
 100% 223/223

S. griseus trypsin (1sgt) S. griseus protease A (2sga)
 $E() < 10^{-14}$ RMSD 1.6 Å $E() < 10^{-4}$; RMSD 2.6 Å
 $E() < 10^{-19}$ 36%; 226/223 $E() < 2.6$ 25%; 199/181

7

When can we infer non-homology?

Non-homologous proteins have different structures

Bovine trypsin (5ptp)
 Structure: $E() < 10^{-23}$,
 RMSD 0.0 Å
 Sequence: $E() < 10^{-84}$
 100% 223/223

Subtilisin (1sbt)
 $E() > 100$
 $E() < 280$; 25% 159/275

Cytochrome c4 (1etp)
 $E() > 100$
 $E() < 5.5$; 23% 171/190

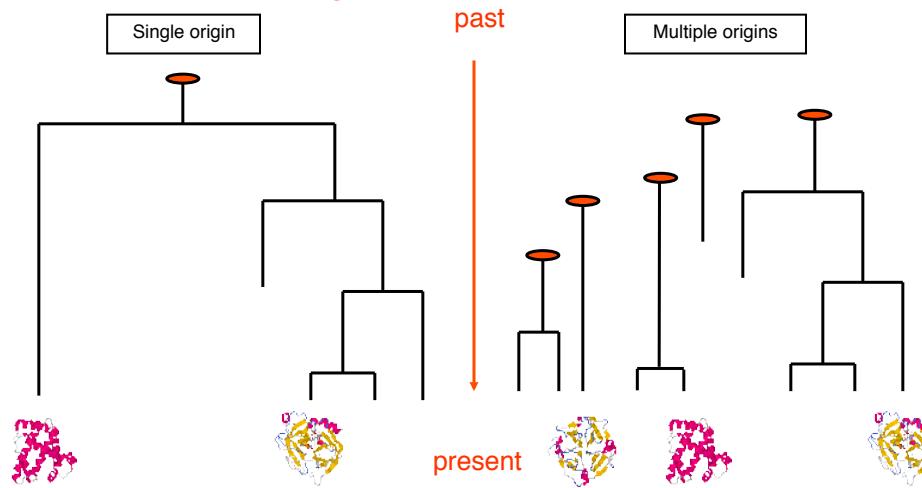
8

Homology is confusing I: Homology defined Three(?) Ways

- Proteins/genes/DNA that share a common ancestor
- Specific positions/columns in a multiple sequence alignment that have a 1:1 relationship over evolutionary history
 - sequences are *50% homologous ???*
- Specific (morphological/functional) characters that share a recent divergence (clade)
 - bird/bat/butterfly wings are/are not homologous

9

Homology is confusing II: Are All Sequences Homologous? **No Homology without excess similarity**



Homology from (sequence/structure) similarity

- Sequences are inferred to share a common ancestor based on statistically significant *excess* similarity. Any evidence of *excess* similarity can be used to infer homology
- Lack of evidence *cannot* be used to infer non-homology.
 - Proteins with different structures are non-homologous
- There are always two alternative hypotheses: homology (common ancestry), or independence – one must weigh the evidence for each hypothesis (independence is the *null* hypothesis).

11

What BLAST does:

Similarity ?
 \Leftrightarrow Homology

Why BLAST works:

Statistical ? Biological
 Significance \Leftrightarrow Significance

Divergence ? Convergence

12

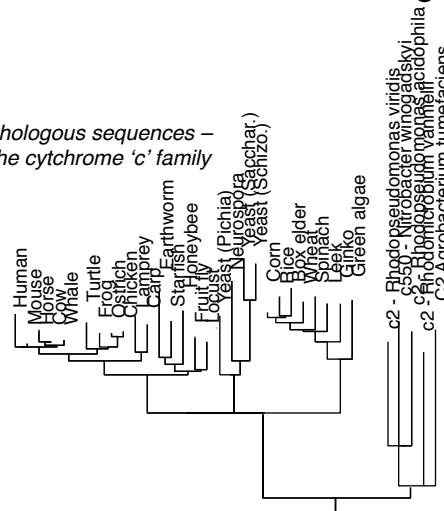
E. coli proteins vs Human – Ancient Protein Domains

expect	% id	alen	E coli descr	Human descr	sp_name
2.7e-206	53.8	944	glycine decarboxylase, P	Glycine dehydrogenase [de	GCSP_HUMAN
1.2e-176	59.5	706	methylmalonyl-CoA mutase	Methylmalonyl-CoA mutase,	MUTA_HUMAN
3.8e-176	50.6	803	glycogen phosphorylase [E	Glycogen phosphorylase, l	PHS1_HUMAN
9.9e-173	55.6	1222	B12-dependent homocysteine	5-methyltetrahydrofolate-	METH_HUMAN
1.8e-165	41.8	1031	carbamoyl-phosphate synth	Carbamoyl-phosphate synth	CPSM_HUMAN
5.6e-159	65.7	542	glucosaphosphate isomeras	Glucose-6-phosphate isome	G6PI_HUMAN
8.1e-143	53.7	855	aconitate hydrase 1 [Esch	Iron-responsive element b	IRE1_HUMAN
2.5e-134	73.0	459	membrane-bound ATP syntha	ATP synthase beta chain,	ATPB_HUMAN
3.3e-121	55.8	550	succinate dehydrogenase,	Succinate dehydrogenase [DHSA_HUMAN
1.5e-113	60.6	401	putative aminotransferase	Cysteine desulfurase, mit	NFS1_HUMAN
4.4e-111	60.9	460	fumarase C= fumarate hydr	Fumarate hydratase, mitoc	FUMH_HUMAN
1.5e-109	56.1	474	succinate-semialdehyde de	Succinate semialdehyde de	SSDH_HUMAN
3.6e-106	44.7	789	maltodextrin phosphorylas	Glycogen phosphorylase, m	PHS2_HUMAN
1.4e-102	53.1	484	NAD+-dependent betaine al	Aldehyde dehydrogenase, E	DHAG_HUMAN
3.8e-98	53.0	449	pyridine nucleotide trans	NAD(P) transhydrogenase,	NNTM_HUMAN
5.8e-96	49.9	489	glycerol kinase [Escheric	Glycerol kinase, testis s	GKP2_HUMAN
2.1e-95	66.8	328	glyceraldehyde-3-phosphat	Glyceraldehyde 3-phosphat	G3P2_HUMAN
5.0e-91	62.5	368	alcohol dehydrogenase cla	Alcohol dehydrogenase cla	ADHX_HUMAN
6.7e-91	56.5	393	protein chain elongation	Elongation factor Tu, mit	EFTU_HUMAN
9.5e-91	56.6	392	protein chain elongation	Elongation factor Tu, mit	EFTU_HUMAN
2.2e-89	59.1	369	methionine adenosyltransf	S-adenosylmethionine synt	METK_HUMAN
6.5e-88	53.3	422	enolase [Escherichia coli	Alpha enolase (2-phospho-	ENO4_HUMAN
9.2e-88	43.3	536	NAD-linked malate dehydro	NADP-dependent malic enzy	MAOX_HUMAN
7.3e-86	55.5	389	2-amino-3-ketobutyrate Co	2-amino-3-ketobutyrate co	KBL_HUMAN
5.2e-83	44.4	543	degrades sigma32, integra	AFG3-like protein 2 (Para	AF32_HUMAN

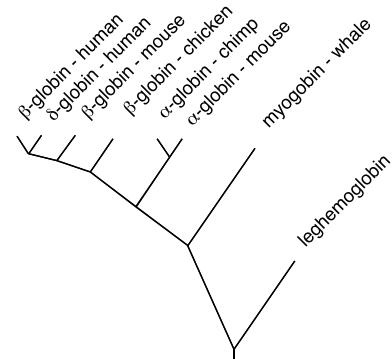
13

Orthologs and Paralogs – Inferring Function

Orthologous sequences – the cytochrome ‘c’ family



Paralogous genes – globins

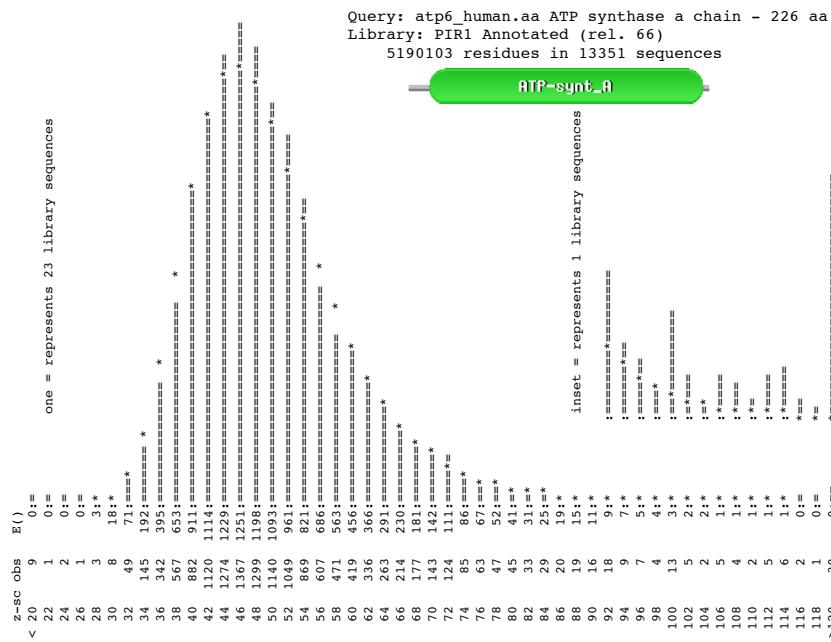


14

Protein Evolution and Sequence Similarity

- What is Homology and how do we recognize it?
- How do we measure sequence similarity – alignments and scoring matrices?
- DNA vs protein comparison
- Alignment Algorithms/Local sequence alignments
- Similarity scoring matrices
- When are we certain that an alignment is significant - similarity score statistics?
- When to trust similarity statistics?
- Improving sensitivity with PSI-BLAST

15



16

Inferring Homology from Statistical Significance

- Real **UNRELATED** sequences have similarity scores that are indistinguishable from **RANDOM** sequences
- If a similarity is NOT **RANDOM**, then it must be NOT **UNRELATED**
- Therefore, NOT **RANDOM** (statistically significant) similarity must reflect **RELATED** sequences

17

Query: atp6_human_aa ATP synthase a chain - 226 aa								
Library: 5190103 residues in 13351 sequences								
The best scores are:								
sp P00846 ATP6_HUMAN	ATP synthase a chain (AT (226)	1400	325.8	5.8e-90	1.000	1.000	226	
sp P00847 ATP6_BOVIN	ATP synthase a chain (AT (226)	1157	270.5	2.5e-73	0.779	0.951	226	
sp P00848 ATP6_MOUSE	ATP synthase a chain (AT (226)	1118	261.7	1.2e-70	0.757	0.916	226	
sp P00849 ATP6_XENLA	ATP synthase a chain (AT (226)	745	176.8	4.0e-45	0.533	0.847	229	
sp P00851 ATP6_DROYA	ATP synthase a chain (AT (224)	473	115.0	1.7e-26	0.378	0.721	222	
sp P00854 ATP6_YEAST	ATP synthase a chain pre (259)	428	104.7	2.3e-23	0.353	0.694	232	
sp P00852 ATP6_EMENI	ATP synthase a chain pre (256)	365	90.4	4.8e-19	0.304	0.691	230	
sp P14862 ATP6_COCH	ATP synthase a chain (AT (257)	353	87.7	3.2e-18	0.313	0.650	214	
sp P05499 ATP6_TOBAC	ATP synthase a chain (AT (395)	309	77.6	5.2e-15	0.283	0.635	233	
sp P68526 ATP6_TRITI	ATP synthase a chain (AT (386)	309	77.6	5.1e-15	0.289	0.651	235	
sp P07925 ATP6_MAIZE	ATP synthase a chain (AT (291)	283	71.7	2.3e-13	0.311	0.667	180	
sp P0AB98 ATP6_ECOLI	ATP synthase a chain (AT (271)	178	47.9	3.2e-06	0.233	0.585	236	
sp POC2Y5 ATPI_ORYSA	Chloroplast ATP synth (A (247)	144	40.1	0.00062	0.242	0.580	231	
sp P06452 ATPI_PEA	Chloroplast ATP synthase a (247)	143	39.9	0.00072	0.250	0.586	232	
sp P27178 ATP6_SYN3	ATP synthase a chain (AT (276)	142	39.7	0.00095	0.265	0.571	170	
sp P06451 ATPI_SPIOL	Chloroplast ATP synthase (247)	138	38.8	0.0016	0.242	0.580	231	
sp P08444 ATP6_SYN6	ATP synthase a chain (AT (261)	127	36.3	0.0095	0.263	0.557	167	
sp P69371 ATPI_ATRBE	Chloroplast ATP synthase (247)	126	36.0	0.01	0.221	0.571	231	
sp P06289 ATPI_MARPO	Chloroplast ATP synthase (248)	126	36.0	0.011	0.240	0.575	167	
sp P30391 ATPI_EUGGR	Chloroplast ATP synthase (251)	123	35.4	0.017	0.257	0.579	214	
sp P19568 TLCA_RICPR	ADP,ATP carrier protein (498)	122	35.0	0.043	0.243	0.579	152	
sp P24966 CYB_TAYTA	Cytochrome b (379)	113	33.0	0.13	0.234	0.532	158	
sp P03892 NU2M_BOVIN	NADH-ubiquinone oxidored (347)	107	31.7	0.31	0.261	0.479	211	
sp P68092 CYB_STEAT	Cytochrome b (379)	104	31.0	0.54	0.277	0.547	137	
sp P03891 NU2M_HUMAN	NADH-ubiquinone oxidored (347)	103	30.8	0.58	0.201	0.537	149	
sp P00156 CYB_HUMAN	Cytochrome b (380)	102	30.5	0.74	0.268	0.585	205	
sp P15993 AR05_ECOLI	Aromatic amino acid tr (457)	103	30.7	0.78	0.234	0.622	111	
sp P24965 CYB_TRAN	Cytochrome b (379)	101	30.3	0.87	0.234	0.563	158	
sp P29631 CYB_POMTE	Cytochrome b (308)	99	29.9	0.95	0.274	0.584	113	
sp P24953 CYB_CAPIH	Cytochrome b (379)	99	29.8	1.2	0.236	0.564	140	

18

ATP-synt_R

```
>>sp|P0AB98|ATP6_ECOLI ATP synthase a chain (ATPase protein 6) g (271 aa)
  s-w opt: 178  Z-score: 218.2  bits: 47.9 E(): 3.2e-06
Smith-Waterman score: 178; 23.3% identity (58.5% similar) in 236 aa overlap (8-222:45-264)

          10      20      30      40      50      60      70      80
human       MNENLFLASPIAPTILGLPAAVLIIILFPPLLPTSKYLIINRLLTQQ
          :...:::..:...:::..:...:::..:...:::..:...:::..:...:::..:
E coli  NMTPQDYGIGHHNNNLQLDLRTFSLVDQPQNPATFWFTINIDSMFFSVVLGL---LFLVLFRSVAKKATSG--VPGKFQTAIE
          10      20      30      40      50      60      70      80

          50      60      70      80      90      100     110
human    WLIKLTSKQMNTKGRTLSMLVSLIIFIATTNLLGLLP-----HSF-----TPTTQLSMNLMAAIPLWAG
  :...:::..:...:::..:...:::..:...:::..:...:::..:...:::..:...:::..:
E coli  LVIGFVNGSVKDHYHGKSKLIALPLALTIFVWWVFLMNLMDLLPIDLLPYIAEHVLGVLPAVRVPSADVNVTLMSALGVF--
          90      100     110     120     130     140     150

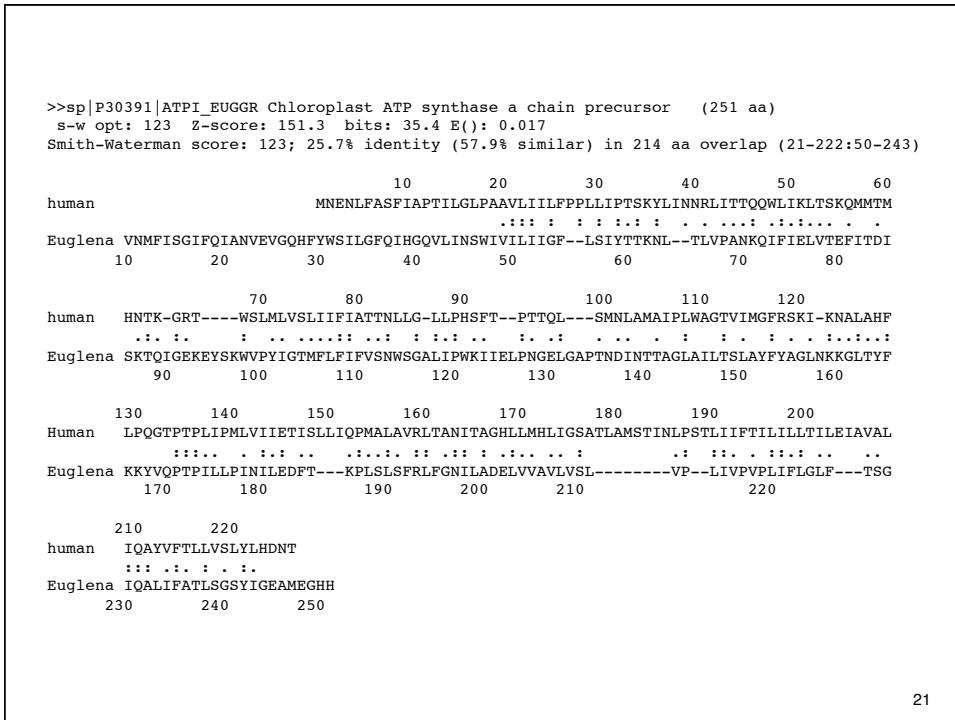
          120     130     140     150     160     170     180
human    TVIMGFRSIKKNALAHFLPQGTPTPL---IPMLVIIETISLLIQPMALAVRLTANITAGHLLMHIGSATLAMSTINL
  ...:::..:...:::..:...:::..:...:::..:...:::..:...:::..:...:::..:...:::..:...:::..:
E coli  -IILIFYSIKMKIGGGFTKELTLQPFNHWAFIGPVNLILEGVSVLLSKPVSLGLRLLFGNMAYAGELIFILIAGLPPWSQWIL
          160     170     180     190     200     210     220     230

          190     200     210     220
human    PSTLILIFTILILITLEIAVALIQAYVFTLLVSLYLHDNT
  ::::..:...:::..:...:::..:...:::..:...:::..:...:::..:
E coli  NVPWAIFHILIT-----LQAFIFMVLTIVYLSMASEEH
          240     250     260     270
```

19

The PAM250 matrix

20



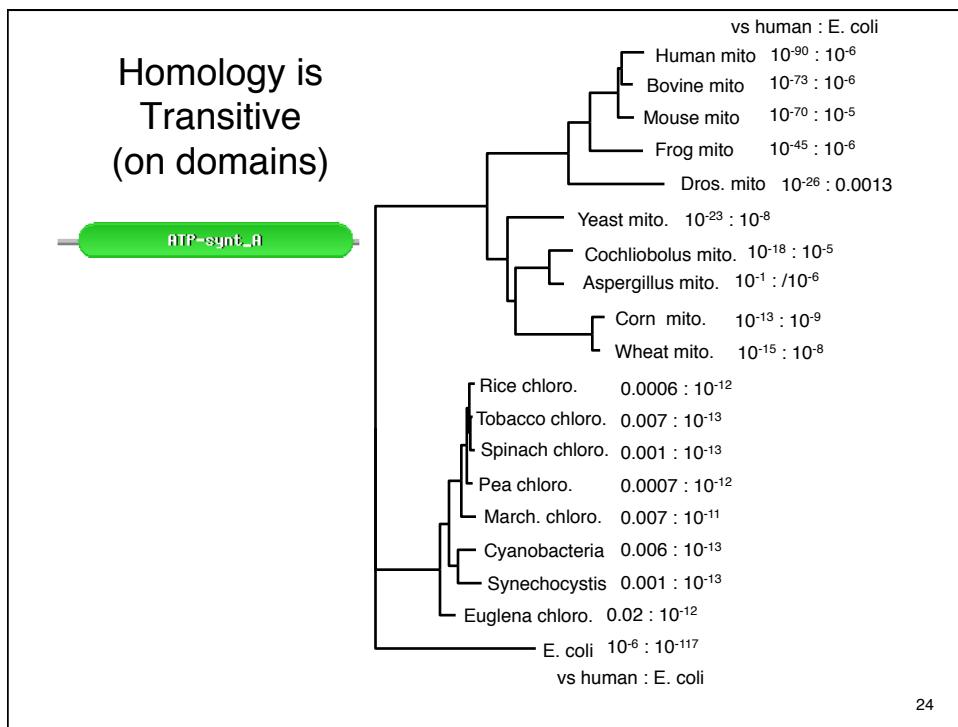
21

Query: atp6_human_aa ATP synthase a chain - 226 aa							
Library: 5190103 residues in 13351 sequences							
The best scores are:							
sp P00846 ATP6_HUMAN ATP synthase a chain (AT (226) 1400 325.8 5.8e-90 1.000 1.000 226	(len)	s-w	bits	E(13351)	%_id	%_sim	alen
sp P00847 ATP6_BOVIN ATP synthase a chain (AT (226) 1157 270.5 2.5e-73 0.779 0.951 226							
sp P00848 ATP6_MOUSE ATP synthase a chain (AT (226) 1118 261.7 1.2e-70 0.757 0.916 226							
sp P00849 ATP6_XENLA ATP synthase a chain (AT (226) 745 176.8 4.0e-45 0.533 0.847 229							
sp P00851 ATP6_DROYA ATP synthase a chain (AT (224) 473 115.0 1.7e-26 0.378 0.721 222							
sp P00854 ATP6_YEAST ATP synthase a chain pre (259) 428 104.7 2.3e-23 0.353 0.694 232							
sp P00852 ATP6_EMENI ATP synthase a chain pre (256) 365 90.4 4.8e-19 0.304 0.691 230							
sp P14862 ATP6_COCHÉ ATP synthase a chain (AT (257) 353 87.7 3.2e-18 0.313 0.650 214							
sp P68526 ATP6_TRITI ATP synthase a chain (AT (386) 309 77.6 5.1e-15 0.289 0.651 235							
sp P05499 ATP6_TOBAC ATP synthase a chain (AT (395) 309 77.6 5.2e-15 0.283 0.635 233							
sp P07925 ATP6_MAIZE ATP synthase a chain (AT (291) 283 71.7 2.3e-13 0.311 0.667 180							
sp P0AB98 ATP6_ECOLI ATP synthase a chain (AT (271) 178 47.9 3.2e-06 0.233 0.585 236							
sp POC225 ATPI_ORYSA Chloroplast ATP synth (A (247) 144 40.1 0.00062 0.242 0.580 231							
sp P06452 ATPI_PEA Chloroplast ATP synthase a (247) 143 39.9 0.00072 0.250 0.586 232							
sp P27178 ATP6_SYN3 ATP synthase a chain (AT (276) 142 39.7 0.00095 0.265 0.571 170							
sp P06451 ATPI_SPIOL Chloroplast ATP synthase (247) 138 38.8 0.0016 0.242 0.580 231							
sp P08444 ATP6_SYN6 ATP synthase a chain (AT (261) 127 36.3 0.0095 0.263 0.557 167							
sp P69371 ATPI_ATRBE Chloroplast ATP synthase (247) 126 36.0 0.01 0.221 0.571 231							
sp P06289 ATPI_MARPO Chloroplast ATP synthase (248) 126 36.0 0.011 0.240 0.575 167							
sp P30391 ATPI_EUGGR Chloroplast ATP synthase (251) 123 35.4 0.017 0.257 0.579 214							
sp P19568 TLCA_RICPR ADP,ATP carrier protein (498) 122 35.0 0.043 0.243 0.579 152							
sp P24966 CYB_TAYTA Cytochrome b (379) 113 33.0 0.13 0.234 0.532 158							
sp P03892 NU2M_BOVIN NADH-ubiquinone oxidored (347) 107 31.7 0.31 0.261 0.479 211							
sp P68092 CYB_STEAT Cytochrome b (379) 104 31.0 0.54 0.277 0.547 137							
sp P03891 NU2M_HUMAN NADH-ubiquinone oxidored (347) 103 30.8 0.58 0.201 0.537 149							
sp P00156 CYB_HUMAN Cytochrome b (380) 102 30.5 0.74 0.268 0.585 205							
sp P15993 AROP_ECOLI Aromatic amino acid tr (457) 103 30.7 0.78 0.234 0.622 111							
sp P24965 CYB_TRAN4 Cytochrome b (379) 101 30.3 0.87 0.234 0.563 158							
sp P29631 CYB_POMTE Cytochrome b (308) 99 29.9 0.95 0.274 0.584 113							
sp P24953 CYB_CAPH1 Cytochrome b (379) 99 29.8 1.2 0.236 0.564 140							

22

Query: atp6_ecoli.aa ATP synthase a - 271 aa Library: 5190103 residues in 13351 sequences						
The best scores are:						
sp POAB98 ATP6_ECOLI ATP synthase a chain (AT (271) 1774 416.8 3.e-117 1.000 1.000 271	(len)	s-w bits	E(13351)	%_id	%_sim	alen
sp P06451 ATPI_SPIOL Chloroplast ATP synthase (247) 274 70.4 5.8e-13 0.270 0.616 211						
sp P69371 ATPI_ATRBE Chloroplast ATP synthase (247) 271 69.7 9.3e-13 0.270 0.607 211						
sp P08444 ATP6_SYNPO ATP synthase a chain (AT (261) 271 69.7 9.9e-13 0.267 0.600 240						
sp P06452 ATPI_PEA Chloroplast ATP synthase a (247) 266 68.5 2.1e-12 0.274 0.614 223						
sp P30391 ATPI_EUGGR Chloroplast ATP synthase (251) 265 68.3 2.5e-12 0.298 0.596 225						
sp P0C2Y5 ATPI_ORYSA Chloroplast ATP synthase (247) 260 67.2 5.4e-12 0.259 0.603 239						
sp P27178 ATP6_SYNY3 ATP synthase a chain (AT (276) 260 67.1 6.1e-12 0.264 0.578 258						
sp P06289 ATPI_MARPO Chloroplast ATP synthase (248) 250 64.8 2.7e-11 0.261 0.621 211						
sp P07925 ATP6_MAIZE ATP synthase a chain (AT (291) 215 56.7 8.7e-09 0.259 0.578 232						
sp P68526 ATP6_TRITI ATP synthase a chain (AT (386) 209 55.3 3.1e-08 0.259 0.603 239						
sp P00854 ATP6_YEAST ATP synthase a chain pre (259) 204 54.2 4.5e-08 0.235 0.578 277						
sp P05499 ATP6_TOBAC ATP synthase a chain (AT (395) 189 50.7 7.8e-07 0.220 0.582 268						
sp P00846 ATP6_HUMAN ATP synthase a chain (AT (226) 178 48.2 2.5e-06 0.237 0.589 236						
sp P00852 ATP6_EMENI ATP synthase a chain pre (256) 178 48.2 2.8e-06 0.209 0.590 244						
sp P00849 ATP6_XENLA ATP synthase a chain (AT (226) 173 47.1 5.5e-06 0.261 0.630 165						
sp P00847 ATP6_BOVIN ATP synthase a chain (AT (226) 172 46.8 6.5e-06 0.233 0.581 236						
sp P14862 ATP6_COCHER ATP synthase a chain (AT (257) 171 46.6 8.7e-06 0.204 0.608 265						
sp P00848 ATP6_MOUSE ATP synthase a chain (AT (226) 166 45.5 1.7e-05 0.259 0.617 193						
sp P00851 ATP6_DROYA ATP synthase a chain (AT (224) 139 39.2 0.0013 0.225 0.549 253						
sp P24962 CYB_STELO Cytochrome b (379) 125 35.9 0.021 0.223 0.575 193						
sp P09716 US17_HCMVA Hypothetical protein HVL (293) 109 32.3 0.21 0.260 0.565 131						
sp P68092 CYB_STEAT Cytochrome b (379) 109 32.2 0.27 0.211 0.562 194						
sp P24960 CYB_ODOHE Cytochrome b (379) 104 31.1 0.61 0.210 0.555 200						
sp P03887 NUIM_BOVIN NADH-ubiquinone oxidoreductase (318) 98 29.7 1.3 0.287 0.545 167						
sp P24992 CYB_ANTAM Cytochrome b (379) 99 29.9 1.4 0.192 0.565 193						

23



24

Homology and Domains – Histone deacetylase PCAF

The best scores are:

	s-w	bits	E(362341)	%_id	%_sim	alen
PCAF_HUMAN Histone acetyltransferase PCAF;	(832)	4876	1092	0	1.000	1.000
PCAF_MOUSE Histone acetyltransferase PCAF;	(813)	4507	1010	0	0.929	0.974
GCNL2_HUMAN General control of amino acid synthesis protein 5-l	(837)	3535	793.	0	0.716	0.864
GCN5_YEAST Histone acetyltransferase GCN5	(439)	1049	240.	5.2e-62	0.469	0.743
GCN5_ARATH Histone acetyltransferase GCN5; AtGCN5	(568)	956	219.	1.2e-55	0.435	0.733
BPTF_HUMAN Nucleosome-remodeling factor subunit BPTF	(3046)	369	88.3	2.4e-15	0.495	0.773
NU301_DROME Nucleosome-remodeling factor subunit NURF301	(2669)	359	86.2	9.3e-15	0.511	0.787
CECR2_HUMAN Cat eye syndrome critical region protein 2	(1484)	306	74.6	1.6e-11	0.371	0.771
BRD4_HUMAN Bromodomain-containing protein 4; HUNK1 protein	(1362)	288	70.6	2.3e-10	0.379	0.681
BRDT_MACFA Bromodomain testis-specific protein	(947)	270	66.7	2.3e-09	0.353	0.690
FSH_DROME Homeotic protein female sterile; Fragile-chorion memb	(2038)	276	67.8	2.4e-09	0.341	0.651
BRDT_HUMAN Bromodomain testis-specific protein; RING3-like prot	(947)	266	65.9	4.3e-09	0.345	0.690
Y0777_DICDI Bromodomain-containing protein DDB_G0280777	(1823)	260	64.3	2.5e-08	0.385	0.725
BRDT_MOUSE Bromodomain testis-specific protein; RING3-like prot	(956)	247	61.6	8.1e-08	0.328	0.647
BAZ2B_HUMAN Bromodomain adjacent to zinc finger domain protein	(1972)	247	61.3	2e-07	0.343	0.695
TAF1_DROME Transcription initiation factor TFIID subunit 1; Tra	(2129)	230	57.5	3.1e-06	0.349	0.689
82_SCHPO Bromodomain-containing protein C631.02	(727)	217	55.0	5.9e-06	0.320	0.587
BRD9_XENLA Bromodomain-containing protein 9	(527)	214	54.5	6.2e-06	0.292	0.579
GTE6_ARATH Transcription factor GTE6; Protein GENERAL TRANSCRIP	(369)	201	51.7	2.9e-05	0.290	0.601
BAZ1B_MOUSE Bromodomain adjacent to zinc finger domain protein	(1479)	212	53.7	3.1e-05	0.302	0.583
K2_SCHPO Bromodomain-containing protein C1450.02	(578)	204	52.2	3.3e-05	0.310	0.628
TAF1_HUMAN Transcription initiation factor TFIID subunit 1; Tra	(1872)	212	53.6	4.2e-05	0.339	0.678
BAZ1B_HUMAN Bromodomain adjacent to zinc finger domain protein	(1483)	209	53.0	5e-05	0.397	0.705
TIF1A_HUMAN Transcription intermediary factor 1-alpha; TIF1-al	(1050)	206	52.5	5.1e-05	0.384	0.698
BDF2_YEAST Bromodomain-containing factor 2	(638)	200	51.3	6.9e-05	0.304	0.607

25

Homology and Domains – Histone deacetylase PCAF

The best scores are:

	E(362341)	alen
PCAF_HUMAN Histone acetyl (832)	0	832
GCN5_YEAST Histone acetyl (439)	5.2e-62	354
BPTF_HUMAN Nucleosome-rem (3046)	2.4e-15	97
CECR2_HUMAN Cat eye syndr (1484)	1.6e-11	105
GTE6_ARATH Transcription (369)	2.9e-05	183

26

Protein Evolution and Sequence Similarity

- What is Homology and how do we recognize it?
- How do we measure sequence similarity – alignments and scoring matrices?
- DNA vs protein comparison
- Alignment Algorithms/Local sequence alignments
- Similarity scoring matrices
- When are we certain that an alignment is significant - similarity score statistics?
- When to trust similarity statistics?
- Improving sensitivity with PSI-BLAST

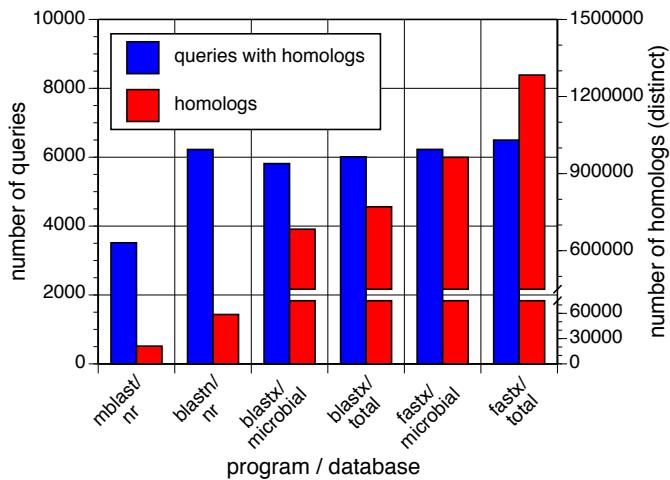
27

DNA vs protein sequence comparison

The best scores are:		DNA E(188,018)	tblastx3 E(187,524)	prot. E(331,956)
DMGST	D.melanogaster GST1-1	1.3e-164	4.1e-109	1.0e-109
MDGST1	M.domestica GST-1 gene	2e-77	3.0e-95	1.9e-76
LUCGLTR	Lucilia cuprina GST	1.5e-72	5.2e-91	3.3e-73
MDGST2A	M.domesticus GST-2 mRNA	9.3e-53	1.4e-77	1.6e-62
MDNF1	M.domestica nf1 gene. 10	4.6e-51	2.8e-77	2.2e-62
MDNF6	M.domestica nf6 gene. 10	2.8e-51	4.2e-77	3.1e-62
MDNF7	M.domestica nf7 gene. 10	6.1e-47	9.2e-77	6.7e-62
AGGST15	A.gambiae GST mRNA	3.1e-58	4.2e-76	4.3e-61
CVU87958	Culicoides GST	1.8e-41	4.0e-73	3.6e-58
AGG3GST11	A.gambiae GST1-1 mRNA	1.5e-46	2.8e-55	1.1e-43
BMO6502	Bombyx mori GST mRNA	1.1e-23	8.8e-50	5.7e-40
AGSUGST12	A.gambiae GST1-1 gene	2.3e-16	4.5e-46	5.1e-37
MOTGLUSTRA	Manduca sexta GST	5.7e-07	2.5e-30	8.0e-25
RLGSTARGN	R.leguminosarum gsta	0.0029	3.2e-13	1.4e-10
HUMGSTT2A	H. sapiens GSTT2	0.32	3.3e-10	2.0e-09
HSGSTT1	H.sapiens GSTT1 mRNA	7.2	8.4e-13	3.6e-10
ECAE000319	E. coli hypothet. prot.	—	4.7e-10	1.1e-09
MYMDCMA	Methyl. dichlorometh. DH	—	1.1e-09	6.9e-07
BCU19883	Burkholderia maleylacetate red.	—	1.2e-09	1.1e-08
NFU43126	Naegleria fowleri GST	—	3.2e-07	0.0056
SP505GST	Sphingomonas paucim.	—	1.8e-06	0.0002
EN1838	H. sapiens maleylaceto. iso.	—	2.1e-06	5.9e-06
HSU86529	Human GSTZ1	—	3.0e-06	8.0e-06
SYCCPNC	Synechocystis GST	—	1.2e-05	9.5e-06
HSEF1GMR	H.sapiens EF1g mRNA	—	9.0e-05	0.00065

28

Improving search strategies (windshield splatter metagenomics)



- always use protein/translated DNA comparisons
 - smaller databases are more sensitive

Similarity Searching II

1. What question to ask?
2. What program to use?
3. What database to search?
4. How to avoid mistakes (what to look out for)
5. When to do something different (changing scoring matrices)

1. What question to ask?

- Is there an homologous protein (a protein with a similar structure)?
- Does that homologous protein have a similar function?
- Does XXX genome have YYY (kinase, GPCR, ...)?

Questions not to ask:

- Does this DNA sequence have a similar regulatory element (too short – never significant)?
- Does (non-significant) protein have a similar function/modification/antigenic site?

31

2. What program to run?

- What is your query sequence?
 - protein – BLAST (NCBI), SSEARCH (EBI)
 - protein coding DNA (EST) – BLASTX (NCBI), FASTX (EBI)
 - DNA (structural RNA, repeat family) – BLASTN (NCBI), FASTA (EBI)
- Does XXX genome have YYY (protein)?
 - TBLASTN YYY vs XXX genome
 - TFASTX YYY vs XXX genome
- Does my protein contain repeated domains?
 - LALIGN (UVA <http://fasta.bioch.virginia.edu>)

32

NCBI BLAST Server

blast.ncbi.nlm.nih.gov

The screenshot shows the NCBI BLAST Server homepage. The title 'NCBI BLAST Server' is on the left, and the URL 'blast.ncbi.nlm.nih.gov' is below it. The main content area is titled 'Basic Local Alignment Search Tool'. It includes a section for 'BLAST Assembled Genomes' with a list of species. Below that is a 'Basic BLAST' section with five options: 'nucleotide blast', 'protein blast', 'blastx', 'tblastx', and 'tblastn'. Each option has a brief description and an 'Algorithms' link. At the bottom is a 'Specialized BLAST' section with four options: 'Make specific primers with Primer-BLAST', 'Search trace archives', 'Find conserved domains in your sequence (cds)', and 'Find sequences with similar conserved domain architecture (cdart)'.

NCBI BLAST Server

blast.ncbi.nlm.nih.gov

Basic BLAST

Choose a BLAST program to run.

nucleotide blast	Search a nucleotide database using a nucleotide query <i>Algorithms: blastn, megablast, discontiguous megablast</i>
protein blast	Search protein database using a protein query <i>Algorithms: blastp, psi-blast, phi-blast</i>
blastx	Search protein database using a translated nucleotide query
tblastx	Search translated nucleotide database using a protein query
tblastn	Search translated nucleotide database using a translated nucleotide query

What is wrong with this picture?

Always compare protein sequences

34

NCBI BLAST Server

BLAST Basic Local Alignment Search Tool

NCBI BLAST/blastp suite

blastn blastp blastx tblastn tblastx

Enter Query Sequence
Enter accession number, gi, or FASTA sequence Clear
Query subrange From To

Or, upload file Choose File no file selected
Job Title
Enter a descriptive title for your BLAST search

Align two or more sequences

Choose Search Set
Database Non-redundant protein sequences (nr)
Organism Optional Enter organism name or id—completions will be suggested Exclude +
Enter organism common name, binomial, or tax id. Only 20 top taxa will be shown.
Entrez Query Optional Enter an Entrez query to limit search

Program Selection
Algorithm blastp (protein-protein BLAST)
 PSI-BLAST (Position-Specific Iterated BLAST)
 PHI-BLAST (Pattern Hit Initiated BLAST)
Choose a BLAST algorithm

BLAST Search database Non-redundant protein sequences (nr) using Blastp (protein-protein BLAST)
 Show results in a new window

► Algorithm parameters

Searching at the EBI

www.ebi.ac.uk/Tools/ssss/

EBI > Tools > Sequence Similarity Searching

Sequence Similarity Searching

BLAST

- NCBI BLAST** NCBI BLAST Sequence Similarity Search using the NCBI BLAST (blastall) program. This tool is available for the following databases:
 - Protein Nucleotide Vectors
- WU-BLAST** Sequence Similarity Search using the Washington University (WU) BLAST2 program (BLAST 2.0 with gaps). This tool is available for the following databases:
 - Protein Nucleotide Parasites
- PSI-BLAST** Position Specific Iterative BLAST (PSI-BLAST) refers to a feature of BLAST 2.0 in which a profile is automatically constructed from the first set of BLAST alignments.
 - Launch PSI-BLAST

FASTA

- FASTA** Sequence Similarity Search using the FASTA program. This tool is available for the following databases:
 - Protein Nucleotide Proteomes Genomes Whole Genome Shotgun
 - ASD Protein ASD Nucleotide LGIC Protein LGIC Nucleotide

SSEARCH

- SSEARCH** Sequence Similarity Search using the SSEARCH program. This tool is available for the following databases:
 - Protein Nucleotide Proteomes Genomes Whole Genome Shotgun
 - ASD Protein ASD Nucleotide LGIC Protein LGIC Nucleotide

PSI-Search

- PSI-Search** PSI-Search combines the sensitivity of the Smith-Waterman search algorithm (SSEARCH) with the PSI-BLAST (blastppg) iterative profile construction strategy to find distantly related protein sequences.
 - Launch PSI-Search

GGSEARCH

- GGSEARCH** GGSEARCH performs a sequence search using alignments that are global in the query and global in the database (Needleman-Wunsch).
 - Protein Nucleotide

36

Searching at the EBI – ssearch

EBI > Tools > Similarity & Homology

FASTA/SSEARCH/GGSEARCH/GLSEARCH - Protein Similarity Search

Provides sequence similarity searching against protein databases using the FASTA and SSEARCH programs. GGSEARCH does a rigorous Smith-Waterman search for similarity between a query sequence and a database. GSEARCH compares a protein or DNA sequence to a sequence database producing global-global alignment (Needleman-Wunsch). GLSEARCH compares a protein or DNA sequence to a sequence database. FASTA can be very specific when identifying long regions of low similarity especially for highly diverged sequences. You can also conduct sequence similarity searching against nucleotide databases or complete proteome/genome databases using the [FASTA programs](#).

[Download Software](#)

PROGRAM	DATABASES	RESULTS	SEARCH TITLE	YOUR EMAIL
SSEARCH	Protein	Interactive	Sequence	
Uniprot Knowledgebase UniProtKB/Swiss-Prot UniProt Clusters 100K UniProt Clusters 100K (SEG filter)				
MATRIX	GAP OPEN	GAP EXTEND	EXPECTATION UPPER VALUE	EXPECTATION LOWER VALUE
BLOSUM50	-10	-2	10.0	default
SCORES	ALIGNMENTS	SEQUENCE RANGE	DATABASE RANGE	FILTER
50	50	START-END	START-END	none
STATISTICAL ESTIMATES				
Regress				
Enter or Paste a PROTEIN Sequence in any format: Help <input type="text"/>				
Upload a file: Choose File no file selected Run Reset				

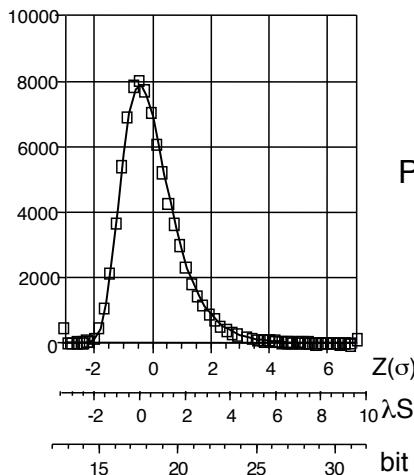
37

3. What database to search?

- Search the smallest comprehensive database likely to contain your protein
 - vertebrates – human proteins (40,000)
 - fungi – S. cerevisiae (6,000)
 - bacteria – E. coli, gram positive, etc. (<100,000)
- Search a richly annotated protein set (SwissProt, 450,000)
- Always search NR (> 12 million) LAST
- Never Search “GenBank” (DNA)

38

Why smaller databases are better – statistics



$$\begin{aligned}
 S' &= \lambda S_{\text{raw}} - \ln K m n \\
 S_{\text{bit}} &= (\lambda S_{\text{raw}} - \ln K) / \ln(2) \\
 P(S' > x) &= 1 - \exp(-e^{-x}) \\
 P(S_{\text{bit}} > x) &= 1 - \exp(-mn2^{-x}) \\
 E(S' > x | ID) &= P D
 \end{aligned}$$

$$\begin{aligned}
 P(B \text{ bits}) &= m n 2^{-B} \\
 P(40 \text{ bits}) &= 1.5 \times 10^{-7} \\
 E(40 | D=4000) &= 6 \times 10^{-4} \\
 E(40 | D=12E6) &= 1.8
 \end{aligned}$$

39

What is a “bit” score?

- Scoring matrices (PAM250, BLOSUM62, VTM40) contain “log-odds” scores:
 $s_{i,j}$ (bits) = $\log_2(q_{i,j}/p_i p_j)$
 $s_{i,j}$ (bits) = 2 \rightarrow a residue is 4-times more likely to occur by homology compared with chance (at one residue)
 $s_{i,j}$ (bits) = -1 \rightarrow a residue is 2-times more likely to occur by chance compared with homology (at one residue)
- An alignment score is the maximum sum of $s_{i,j}$ bit scores across the aligned residues. A 40-bit score is 2^{40} more likely to occur by homology.
- How often should a score occur by chance? In a 400 * 400 alignment, there are 160,000 places where the alignment could start by chance, so we expect a score of 40 bits would occur:
 $400 \times 400 \times 2^{40} = 1.6 \times 10^5 / 2^{40} (10^{13.3}) = 0.8 \times 10^{-8}$ times
 Thus, the probability of a 40 bit score in ONE alignment is $\sim 10^{-8}$
- But we did not ONE alignment, we did 4,000, 40,000, 400,000, or 16 million):
 $E(p | D) = p(40 \text{ bits}) \times \text{database size}$
 $E(40 | 4,000) = 10^{-8} \times 4,000 = 4 \times 10^{-5}$ (significant)
 $E(40 | 40,000) = 10^{-8} \times 4 \times 10^4 = 4 \times 10^{-4}$ (significant)
 $E(40 | 400,000) = 10^{-8} \times 4 \times 10^5 = 4 \times 10^{-3}$ (not significant)

40

How many “bits” do I need?

$$E(p | D) = p(40 \text{ bits}) \times \text{database size}$$

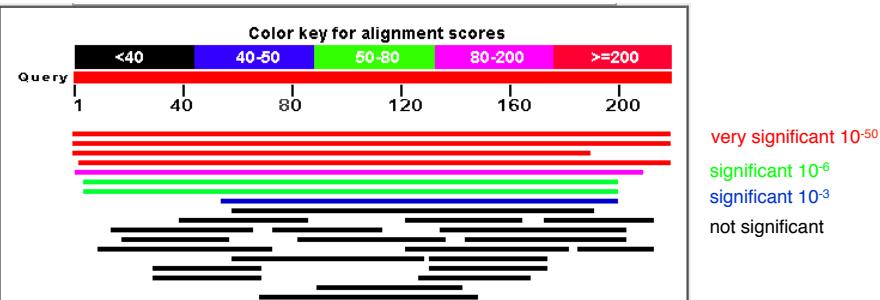
$E(40 4,000) = 10^{-8} \times 4,000 = 4 \times 10^{-5}$		(significant)
$E(40 40,000) = 10^{-8} \times 4 \times 10^4 = 4 \times 10^{-4}$		(significant)
$E(40 400,000) = 10^{-8} \times 4 \times 10^5 = 4 \times 10^{-3}$		(not significant)

To get $E() \sim 10^{-3}$:

genome (10,000) $p \sim 10^{-3}/10^4 = 10^{-7}/160,000 = 40$ bits

SwissProt (500,000) $p \sim 10^{-3}/10^6 = 10^{-9}/160,000 = 47$ bits

Uniprot/NR (10^7) $p \sim 10^{-3}/10^7 = 10^{-10}/160,000 = 50$ bits



41

Similarity Searching II

1. What question to ask?
2. What program to use?
3. What database to search?
4. How to avoid mistakes (what to look out for)
5. When to do something different

42

Inferring Homology from Statistical Significance

- Real ***UNRELATED*** sequences have similarity scores that are indistinguishable from ***RANDOM*** sequences
- If a similarity is NOT ***RANDOM***, then it must be NOT ***UNRELATED***
- Therefore, NOT ***RANDOM*** (statistically significant) similarity must reflect ***RELATED*** sequences

43

Smith-Waterman (sssearch)

		s-w	bits	E(115640)	%_id	alen
GTM1_MOUSE	Glutathione S-trans	(218)	1497	363.5	2e-100	1.000 218
GTM2_CHICK	Glutathione S-trans	(220)	958	234.9	1.e-61	0.619 218
GTP_HUMAN	Glutathione S-trans	(210)	356	91.2	1.8e-18	0.308 211
PGD2_MOUSE	Glutathione-req.	(199)	262	68.8	9.7e-12	0.319 204
GTA1_MOUSE	Glutathione S-trans	(223)	229	60.9	2.6e-09	0.284 225
SC1_OCTDO	S-crystallin 1 OLL	(215)	228	60.7	3.0e-09	0.269 219
GTS_MUSDO	Glutathione S-trans	(241)	228	60.6	3.4e-09	0.264 201
GTS1_CAEEL	Prob. Glut. S-trans	(210)	220	58.8	1.1e-08	0.284 225
GTS_OMMSL	Glutathione S-trans	(203)	196	53.0	5.5e-07	0.258 209
GTH3_ARATH	Glutathione S-trans	(215)	142	40.1	0.0045	0.310 126
GTT2_HUMAN	Glutathione S-trans	(244)	132	37.7	0.027	0.257 167
GT24_DROME	Glutathione S-trans	(216)	131	37.5	0.028	0.255 153
YFCG_ECOLI	Hypothetical GST	(215)	112	33.0	0.64	0.235 187
YJY1_YEAST	hypothetical	30.5	(261)	110	32.4	*1.1*
DCMA_METS1	dichloromethane DM	(267)	103	30.8	3.7	0.214
YA42_HAEIN	Hypothetical prot.	(617)	108	31.7	*4.6*	0.283
GTO1_RAT	Glutathione trans	(241)	100	30.1	5.4	0.234
DP41_BACHD	DNA polymerase I	(413)	104	30.8	*5.4*	0.234
GTH1_WHEAT	Glutathione S-trans	(229)	98	29.6	7.0	0.246
LGUL_SOYBN	Lactoylglutathione	(219)	97	29.4	7.8	0.200

Highest scoring unrelated sequence E() ~ 1.0

44

Unrelated ≠ Random (low complexity)

Search with complete grou_drome:

The best scores are:

	opt	bits	E(14548)
RGHUB1 GTP-binding regulatory protein beta-1 chai (341)	237	46.6	3.5e-05
RGBOB1 GTP-binding regulatory protein beta-1 chai (341)	237	46.6	3.5e-05
RGHUB3 GTP-binding regulatory protein beta-3 chai (341)	233	46.0	5.2e-05
RGMSB4 GTP-binding regulatory protein beta-4 chai (341)	232	45.8	5.7e-05
PIHUPF salivary proline-rich glycoprotein precurs (252)	224	44.5	*0.00010*
RGFFB GTP-binding regulatory protein beta chain (347)	223	44.5	0.00014
PIRT3 acidic proline-rich protein precursor - rat (207)	199	40.8	*0.0011*
PIHUB6 salivary proline-rich protein precursor PR (393)	203	41.6	*0.0012*
CGB02S collagen alpha 2(I) chain - bovine (fragme (403)	195	40.5	*0.0027*
WMBEW6 capsid protein - human herpesvirus 1 (stra (636)	192	40.2	*0.0051*

Search with seg-ed grou_drome: (low complexity regions removed)

The best scores are:

	opt	bits	E(14548)
RGHUB3 GTP-binding regulatory protein beta-3 chai (341)	233	56.5	3.6e-08
RGMSB4 GTP-binding regulatory protein beta-4 chai (341)	232	56.3	4.1e-08
RGHUB2 GTP-binding regulatory protein beta-2 chai (341)	228	55.5	7.2e-08
RGBOB1 GTP-binding regulatory protein beta-1 chai (341)	225	54.9	1.1e-07
RGFFB GTP-binding regulatory protein beta chain (347)	223	54.5	1.5e-07
BVBYMS MSI1 protein - yeast (Saccharomyces cerevi (423)	135	37.0	*0.033*
ERHUAH coatomer complex alpha chain homolog - hum (1225)	134	37.1	*0.088*
A28468 chromogranin A precursor - human (458)	122	34.4	*0.21*
RGOOBE GTP-binding regulatory protein beta chain (342)	120	33.9	0.22
			45

pseg removes low-complexity regions

>gi|17380405|sp|P16371|GROU_DROME Groucho protein (Enhancer of split M9/10)

paagggppppqgp	1-8	MYPSPVRH
	9-19	
	20-131	IKFTIADTLERIKEEFNFLQAQYHSIKLEC EKLSNEKTEMQRHYVMMYEMSYGLNVEMHK QTEIAKRLNLTINQLLPFLQADHQQVLQA VERAKQVTMQEELNLLIIGQQIHA
qqvpvgggpppqmg	132-143	
	144-281	ALNPFGALGATMGLPHGPQGLLNKPPEHHR PD1KPTGLEGPAAAERRLRNSVSPADREKY RTRSPLDIENDSKRRKDEKLQDEGEKSDQ DLVVVDVANEAMESHSPRPNGEHVSMEVRDRE SLNGERLEKPSSSGIKQE
rppssrsgssssrstps	282-297	
	298-310	LTKTKDMEKPGTPG
akartptpnaaaapgvnpk	311-330	
qmmppqgpppagypgapyqrpa	331-351	
	352-719	DPYQRPPSDPAYGRPPPMYDPHABVRTNG IPHPHSALTGGKFAYFSFHMNMGEGSLQPVPFP PDALVGVGIPRHRQINTLSHGEVVCATTI SNPTKVYYGGKGCVKVVWDISOPGNKNPVS QLDCLQRDNYIIRSVKLLPDGRTLIVGGEAS NLISIWDLASPTPRIKAELTSAAAPACYALAI SPDSKVCFCSCSDGNIAVWDLHNEILVRQF QGHTDGASCID1ISPDCGSRLWTGGLDNTVRS WDLREGROLQQHDFSSQIFSLGYCPTGDWL AVGMENSHVEVLHASPKPDKYQLHLHESCVL SLRFAACGKWFVSTGKDNLNAWRTPY GAS IFQSKEETSSVLCSDISTDDKYIVTGSSDKK ATVYEVII

46

Validating homologs/statistics

- In general, BLASTP statistical estimates are accurate
- The most common errors occur because of low-complexity regions, or biased amino-acid composition
- To confirm statistical accuracy, find the highest scoring non homolog
 - No need to test every hit, test hits that are surprising
 - Confirm homology/non-homology by searching against a different comprehensive database, e.g. SwissProt, or refseq.
 - Non-homologs will find many significant members of other families, but not the family you are testing for
- Statistical estimates can be confirmed with shuffles (see ISMB2000 tutorial, [fasta.bioch.virginia.edu/fasta_www2/shuffle link](http://fasta.bioch.virginia.edu/fasta_www2/shuffle.html))

47

Scoring matrices

- Scoring matrices can set the evolutionary look-back time for a search
 - Lower PAM (PAM10/MDM10 ... PAM60) for closer (10% ... 50% identity)
 - Higher BLOSUM for higher conservation (BLOSUM50 distant, BLOSUM80 conserved)
- Shallow scoring matrices for short domains/short queries (metagenomics)
 - Matrices have “bits/position” (score/position), 40 aa at 0.7 bits/position (BLOSUM62) means 28 bit max score (50 bits significant)
- Deep scoring matrices allow alignments to continue, possibly outside the homologous region

48

Finding Domains – Local alignments: calmodulin

```

qRegion: 8-43:84-116 : score=102; bits=25.6; Q=33.5 : DOMAIN_N: EF-hand 1.
qRegion: 44-76:117-149 : score=101; bits=25.4; Q=32.9 : DOMAIN_N: EF-hand 2.
Waterman-Eggert score: 220; 48.7 bits; E(1) < 5e-11
46.1% identity (73.7% similar) in 76 aa overlap (1-76:77-149)
[10 20 30 40 ][ 50 60 70
sp|P62 MADQLTEEQIAEKFKEAFLFDKDGDTITTKELGTVMRSLGQNPTAEALQDMINEVDADGNNTIDFPEFLTMMARK
: : .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .
sp|P62 MKDTDSEEEEI---REAFRVFDKDNGYISAAELRHVMTNLGEKLTDDEEVDEMIREADIDGDGQVNYYEEFVQMMTAK
80 90 100 110 120 130 140
-----
qRegion: 11-43:47-80 : score=61; bits=17.5; Q=9.6 : DOMAIN_N: EF-hand 1.
qRegion: 44-79:81-119 : score=62; bits=17.7; Q=10.1 : DOMAIN_N: EF-hand 2.
qRegion: 81-111:121-147 : score=59; bits=17.1; Q=8.5 : DOMAIN_N: EF-hand 3.
Waterman-Eggert score: 181; 41.0 bits; E(1) < 9.9e-09
34.3% identity (64.8% similar) in 105 aa overlap (11-111:47-147)
20 30 40 ][ 50 60 70 10[ 120
sp|P62 AEFKEAFLFDKDGDTITTKELGTVM-RSLQNPTEAEALQDMINEVDADGNNTIDFPEF---LTMMARKMKTDSEEEEI
: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .
sp|P62 AEQDMINEVDADGNNTIDFPEFLTMMARKMKTDSEEEEIREAFRVFKDNGYISAAELRHVMTNLGEKLTDDEEVDEMI
50 60 70 80 90 100 110 120
sp|P62 REAFRVFDKDNGYISAAELRHVMT
::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .
sp|P62 REA---DIDGDGQVNYYEEFVQMMT
130 140
-----
qRegion: 8-37:120-146 : score=45; bits=14.4; Q=1.9 : DOMAIN_N: EF-hand 1.
Waterman-Eggert score: 64; 18.1 bits; E(1) < 0.075
34.2% identity (71.1% similar) in 38 aa overlap (1-37:113-146)
[10 20 30
sp|P62 MADQLTEQIAEKFKEAFLFDKDGDTITTKELGTVM
.::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .::: .
sp|P62 LGEKLTDDEEVDEMIREA---DIDGDGQVNYYEEFVQMM
120 130 140

```

49

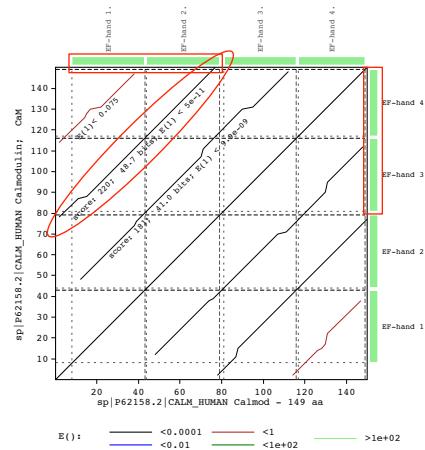
Repeated domains with local alignments

```

EF-hand 1: 8-43: 84-116:s=102; bits=25.6; Q=33.5
EF-hand 2: 44-76:117-149:s=101; bits=25.4; Q=32.9

Waterman-Eggert score: 220; 48.7 bits; E(1)< 5e-11
46.1% identity in 76 aa overlap (1-76:77-149)
[10 20 30 40
sp|P62 MADQLTEEQIAEKFKEAFLFDKDGDTITTKELGTVMRSL
: : .::: .::: .::: .::: .::: .::: .::: .::: .::: .
sp|P62 GQNPTAEALQDMINEVDADGNNTIDFPEFLTMMARK
: : .::: .::: .::: .::: .::: .::: .::: .::: .::: .
sp|P62 GEKLTDEEVDEMIREADIDGDGQVNYYEEFVQMMTAK
120 130 140

```



50

More about scoring matrices ...

PAM series:

- Evolutionary model - extrapolated from PAM1
- PAM20: 20% change (mammals)
- PAM250: 250% change (<20% identity)
- Gap penalties should vary
- shallow matrices (PAM10-40) for short sequences and short distances

BLOSUM series

- Empirically determined, no extrapolation (no model)
- BLOSUM45-50 - distant (1/3 bits)
- BLOSUM80 -very highly conserved (not small change), high info/position
- BLOSUM62 - 1/2 bits

51

Where do scoring matrices come from?

Pam40

A	R	N	D	E	I	L
A	8					
R	-9	12				
N	-4	-7	11			
D	-4	-13	3	11		
E	-3	-11	-2	4	11	
I	-6	-7	-7	-10	-7	12
L	-8	-11	-9	-16	-12	-1

Pam250

A	R	N	D	E	I	L
A	2					
R	-2	6				
N	0	0	2			
D	0	-1	2	4		
E	0	-1	1	3	4	
I	-1	-2	-2	-2	-2	5
L	-2	-3	-3	-4	-3	2

q_{ij} : replacement frequency at PAM40, 250

$$q_{R:N(40)} = 0.000435$$

$$q_{R:N(250)} = 0.002193$$

$$p_R = 0.051$$

$$p_N = 0.043$$

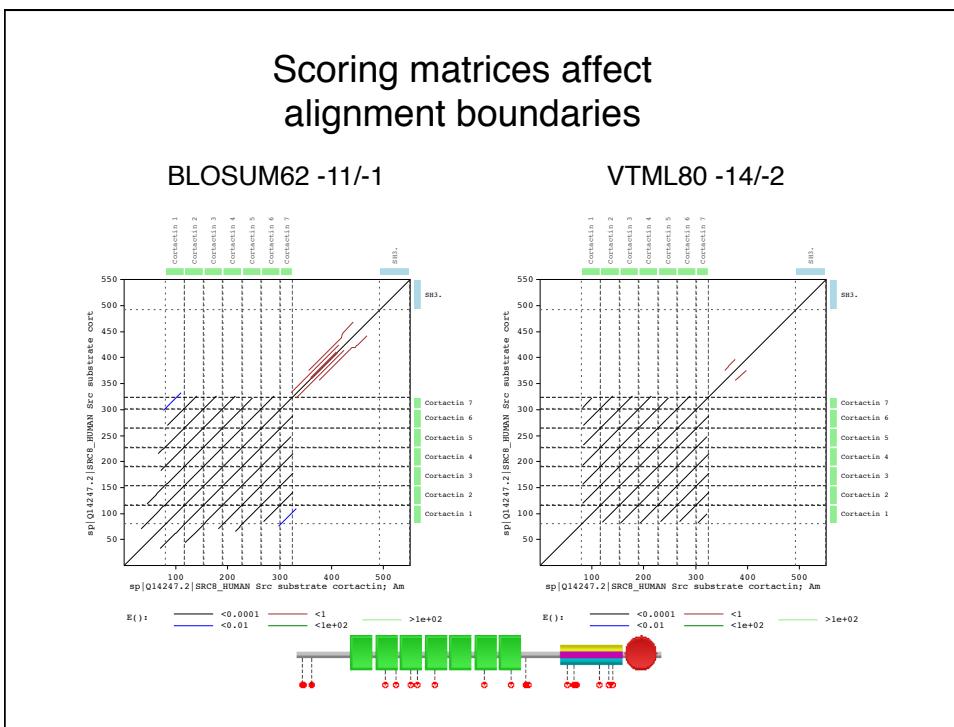
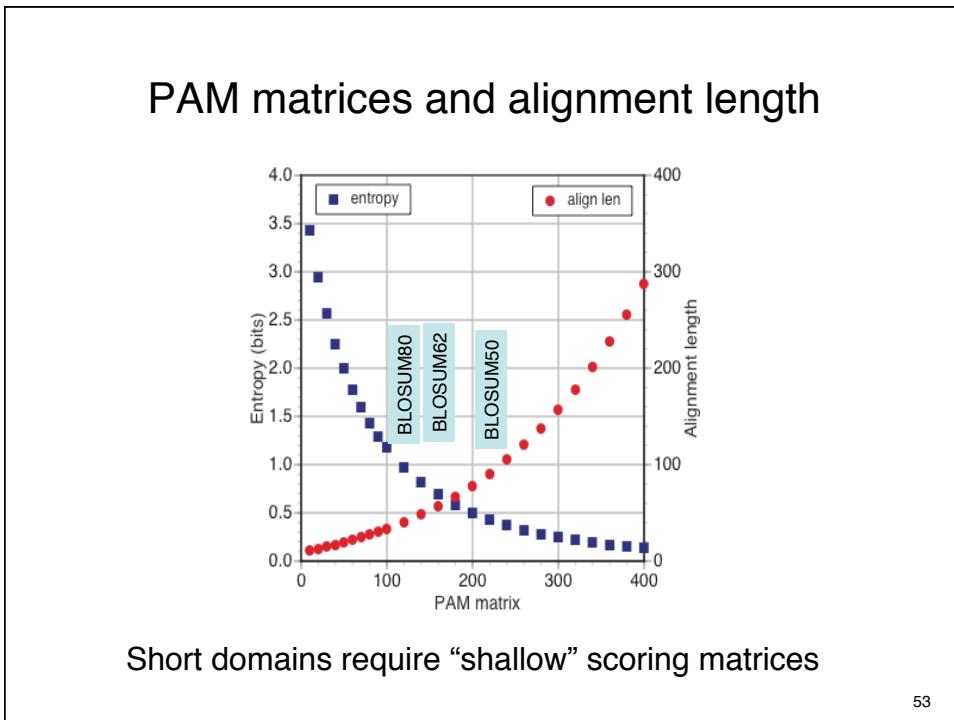
$$I_2 S_{ij} = \lg_2 (q_{ij}/p_i p_j) \quad I_e S_{ij} = \ln(q_{ij}/p_i p_j) \quad p_R p_N = 0.002193$$

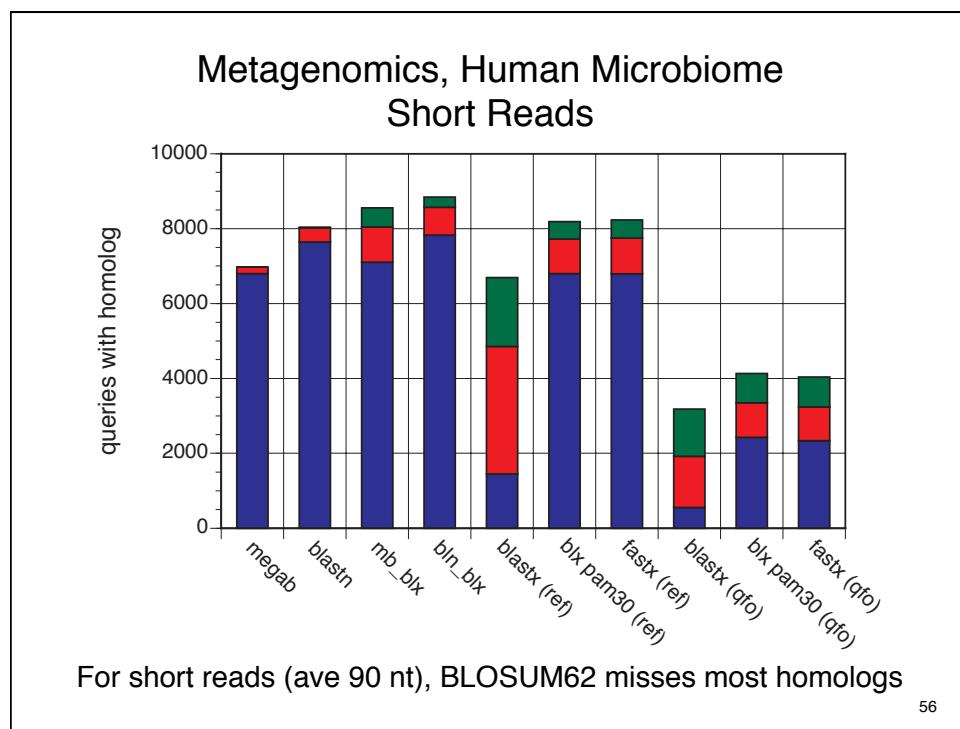
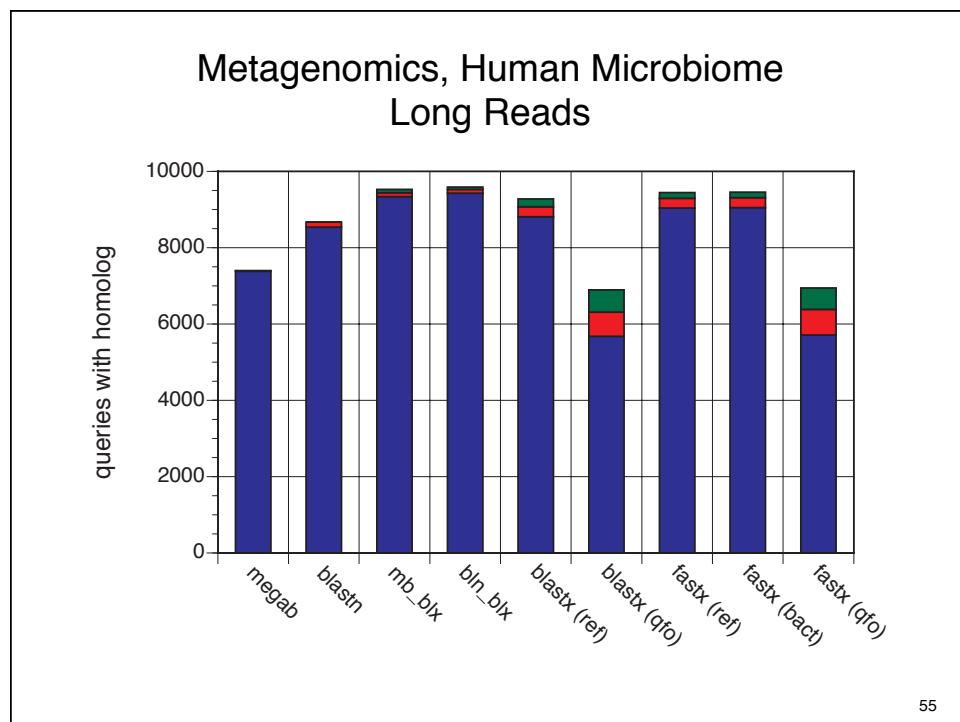
$$I_2 S_{R:N(40)} = \lg_2 (0.000435/0.002193) = -2.333$$

$$I_2 = 1/3; S_{R:N(40)} = -2.333/I_2 = -7$$

$$I_2 S_{R:N(250)} = \lg_2 (0.002193/0.002193) = 0$$

52





Scoring Matrices - Summary

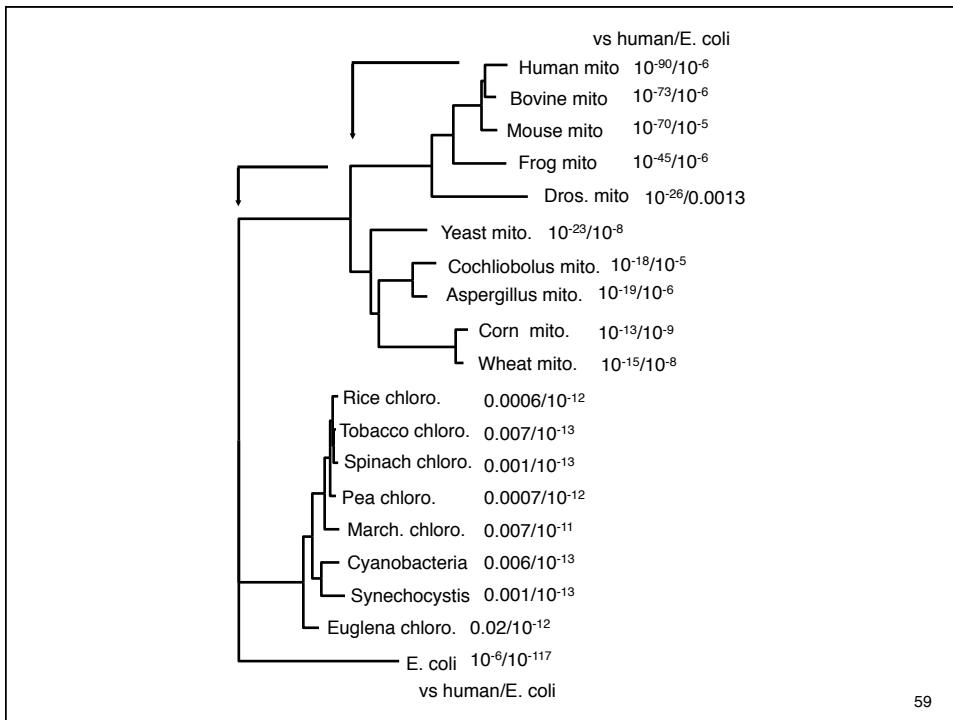
- PAM and BLOSUM matrices greatly improve the sensitivity of protein sequence comparison – low identity with significant similarity
- PAM matrices have an evolutionary model - lower number, less divergence – lower=closer; higher=more distant
- BLOSUM matrices are sampled from conserved regions at different average identity – higher=more conservation
- Short alignments require shallow matrices
- Shallow matrices set maximum look-back time

57

Similarity Searching II

1. What question to ask?
2. What program to use?
3. What database to search?
4. How to avoid mistakes (what to look out for)
5. When to do something different
6. PSI-BLAST – the most sensitive method

58



ATP synthase - matrices, gaps, algorithms

Matrix: Gap open/extend	BLOSUM50 -10/-2	BLOSUM62 -11/-1	BLASTP -11/-1
The best scores are:			
ATP6_HUMAN ATP synthase a chai	297.7 1.7e-81	373.6 2.4e-104	296 3e-81
ATP6_BOVIN ATP synthase a chai	252.4 7.2e-68	310.7 2.0e-85	253 2e-68
ATP6_MOUSE ATP synthase a chai	246.4 4.5e-66	302.9 4.4e-83	245 5e-66
ATP6_XENLA ATP synthase a chai	111.9 1.4e-25	125.9 8.7e-30	142 9e-35
ATP6_YEAST ATP synthase a ch	78.7 1.6e-15	90.1 5.7e-19	93 5e-20
ATP6_EMENI ATP synthase a chai	66.3 8.4e-12	76.6 6.8e-15	75 2e-14
ATP6_DROYA ATP synthase a chai	65.6 1.2e-11	75.4 1.4e-14	101 2e-22
ATP6_COCHET ATP synthase a cha	53.6 5.5e-08	60.6 4.6e-10	75 1e-14
ATP6_ECOLI ATP synthase a ch	45.1 2.2e-05	49.1 1.4e-06	42 1e-04
ATP6_TRITI ATP synthase a ch	45.0 3.3e-05	50.7 6.5e-07	83 5e-17
ATP6_TOBAC ATP synthase a chai	40.4 0.00084	47.0 8.6e-06	80 3e-16
ATP6_MAIZE ATP synthase a chai	39.6 0.001	44.9 2.6e-05	
ATPI_PEA Chloroplast ATP syn	35.8 0.013	38.0 0.0028	
ATPI_SPIOL Chloroplast ATP syn	35.5 0.015	38.0 0.0028	
ATPI_ATRBE Chloroplast ATP s	34.0 0.044	36.3 0.0086	
ATPI_MARPO Chloroplast ATP syn	33.2 0.075	34.3 0.036	
HBA_ODOVI Hemoglobin subunit a		31.9 0.11	
*AROP_ECOLI Aromatic amino ac	32.1 0.31	31.4 0.5 *	
ATPI_EUGGR Chloroplast ATP syn	31.1 0.32	32.2 0.15	
ATP6_SYNPG6 ATP synthase a chai	31.1 0.34	31.8 0.21	
TLCR_RICPR ADP,ATP carrier pro	31.5 0.49	29.7 1.7	
ATP6_SYNY3 ATP synthase a chai	30.6 0.51	31.8 0.22	28 1.9
ATPI_ORYSA Chloroplast ATP	30.1 0.65	32.2 0.15	
GLUC_MYOSC Glucagon precursor	28.7 0.65	34.4 0.013	
VP6_BPPH6 Protein P6	29.1 0.85	28.6 1.3	
GLUC_LEPSP Glucagon precursor	27.7 1.	32.7 0.033	
ADH1_MOUSE Alcohol dehydrogena	29.8 1.2	34.4 0.013	

60

Metazoan ATP Synthases

CLUSTAL W (1.81) multiple sequence alignment

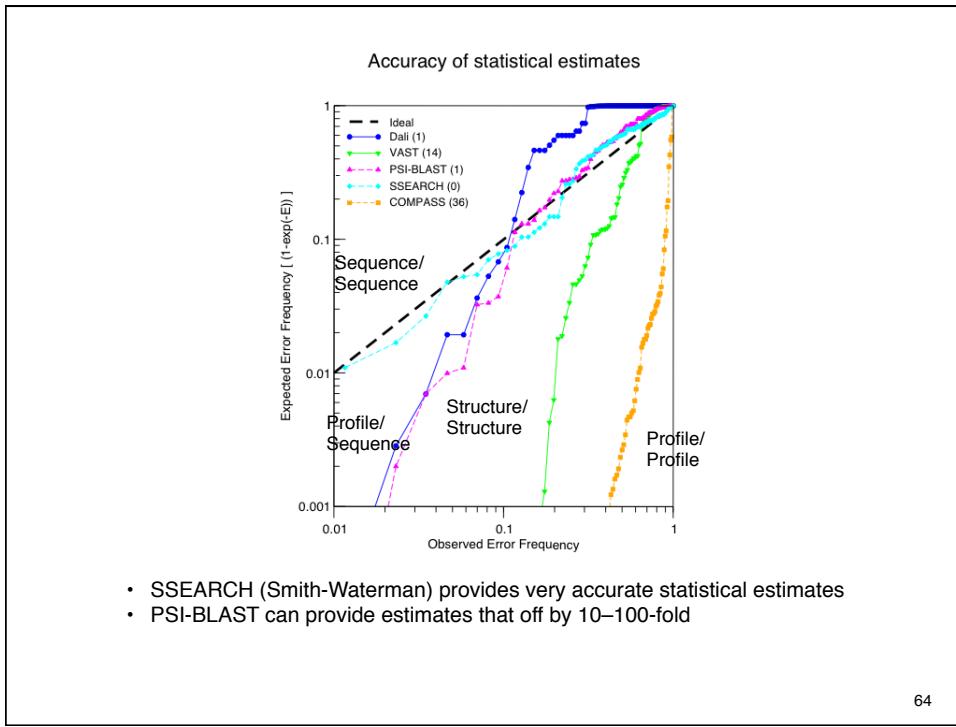
61

PSI-BLAST ATP6_HUMAN - 4 iterations

Results from round:		(1)		(2)		(3)		(4)		
Sequences producing significant alignments:		Score (bits)	E Value	Score (bits)	E Value	Score (bits)	E Value	Score (bits)	E Value	
ATP6_HUMAN	ATP synthase a chain (ATPase protein 6)	296	3e-81	257	1e-69	241	2e-62	222	5e-59	
ATP6_BOVIN	ATP synthase a chain (ATPase protein 6)	253	2e-68	257	2e-69	239	8e-65	230	2e-61	
ATP6_MOUSE	ATP synthase a chain (ATPase protein 6)	245	5e-66	247	3e-66	234	4e-64	225	6e-60	
ATP6_XENLA	ATP synthase a chain (ATPase protein 6)	142	9e-35	227	1e-60	189	3e-49	177	2e-45	
ATP6_DROYA	ATP synthase a chain (ATPase protein 6)	101	2e-22	206	3e-54	209	5e-55	196	4e-51	
(2)										
ATP6_YEAST	ATP synthase a chain precursor (ATPase prot	93	5e-20	97	3e-21	199	4e-52	191	2e-49	
ATP6_TRITI	ATP synthase a chain (ATPase protein 6)	83	5e-17	96	5e-21	218	1e-57	236	4e-63	
(3)										
ATP6_TOBAC	ATP synthase a chain (ATPase protein 6)	80	3e-16	90	4e-19	200	2e-52	230	3e-61	
ATP6_MAIZE	ATP synthase a chain (ATPase protein 6)	76	5e-15	88	1e-18	198	1e-51	219	5e-58	
ATP6_COCHET	ATP synthase a chain (ATPase protein 6)	75	1e-14	86	9e-18			197	2e-51	
ATP6_EMENI	ATP synthase a chain precursor (ATPase prot	75	2e-14	84	3e-17	123	5e-29	181	2e-46	
(4)										
ATP6_ECOLI	ATP synthase a chain (ATPase protein 6)	42	1e-04	40	5e-04	46	8e-06	49	1e-06	
ATPI_SPIOL	Chloroplast ATP synthase a chain precursor			32	0.12	36	0.006	39	0.001	
ATP6_SYNY3	ATP synthase a chain (ATPase protein 6)	28	1.9	32	0.16	44	5e-05	45	1e-05	
ATPI_MARPO	Chloroplast ATP synthase a chain precursor			31	0.21	44	4e-05	44	3e-05	
ATPI_PEA	Chloroplast ATP synthase a chain precursor (A			31	0.32	37	0.005			
LAMA2_MOUSE	Laminin subunit alpha-2 precursor (Laminin			31	0.34					
ATPI_ATRBE	Chloroplast ATP synthase a chain precursor			31	0.39	41	2e-04			
ATP6_SYNP6	ATP synthase a chain (ATPase protein 6)			28	1.7	41	2e-04			
ATPI_EUGCR	Chloroplast ATP synthase a chain precursor							39	0.001	
ATPI_ORYSA	Chloroplast ATP synthase a chain precursor							28	1.9	
ATPI_ATRBE	Chloroplast ATP synthase a chain precursor							36	0.009	
ATP6_ASAPM	ATP synthase a chain (ATPase protein 6)								36	0.008
POLG_KUNJIN	Genome polyprotein (Contains: Capsid protei...	27	5.0							
POL_HTLIC	Gag-Pro-Pol polyprotein (Pr160Gag-Pro-Pol) [...	27	5.0							
POLG_DENZU2	Genome polyprotein (Contains: Capsid protei...	27	5.2	26	7.0					

62

		Position-Specific Scores ATP Synthase, 4 iterations																				
		A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	bits/pos
BL62	Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0.70
46	Q	-2	-1	-2	-2	-4	6	0	1	0	-4	-3	-1	-2	-1	-3	-1	-2	6	4	-3	0.74
47	Q	-1	-1	3	3	-3	3	3	-2	3	-4	-4	-1	-3	-4	-2	2	-1	-4	-2	-3	0.51
56	Q	-2	-1	-2	-2	-3	5	2	-4	-1	4	-1	-1	-1	-2	-3	-2	-2	-3	-2	0	0.51
97	Q	-2	-1	0	-2	-4	4	0	-3	8	-4	-4	-1	-2	-3	-3	-1	-2	-3	0	-4	1.11
131	Q	3	-1	-1	-1	-2	5	2	-2	-1	-3	-3	0	-2	-4	-2	1	-1	-3	-3	-2	0.52
152	Q	-2	6	-1	-2	-4	4	0	-3	-1	-4	-3	1	-2	-4	-3	-1	-2	-4	-3	-3	1.00
210	Q	-2	0	-1	-1	-4	7	1	-3	0	-4	-3	1	-1	-4	-2	-1	-2	-3	-2	-3	1.13
	%	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	



Why does PSI-BLAST fail?

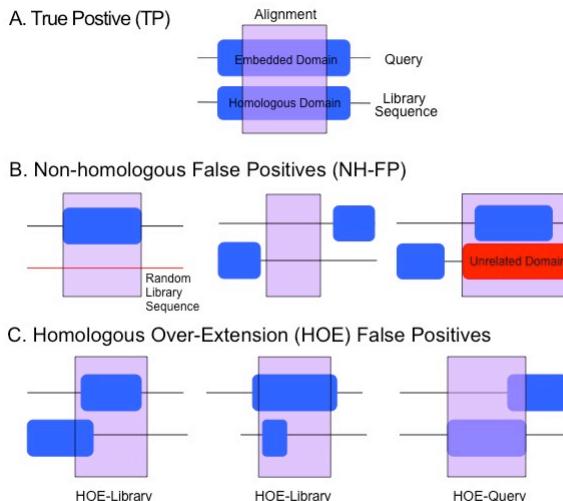


Figure 1

Sensitive searches with PSI-BLAST

- PSI-BLAST improves sensitivity by building a Position Specific Scoring Matrix (PSSM)
 - models ancestral sequence (consensus distribution)
 - similar to PFAM HMM (but less sophisticated weights, gaps)
- Sensitivity improves with additional iterations
 - model moves to base of tree
- Statistical estimates are difficult
 - once a sequence is in, it is “significant” - validation must be done before a sequence is included
- Very diverse families may not produce a well defined PSSM
 - similar problems with HMMs have led to “clans”

66

Sequence Similarity - Conclusions

- Homologous sequences share a common ancestor, but most sequences are non-homologous
- Always compare Protein Sequences
- Sequence Homology can be reliably inferred from statistically significant similarity (non-homology cannot from non-similarity)
- Homologous proteins share common structures, but not necessarily common functions
- Sequence statistical significance estimates are accurate (verify this yourself) $10^{-6} < E() < 10^{-3}$ is statistically significant
- Scoring matrices set evolutionary look back horizons - not every discovery is distant
- PSI-BLAST can be more sensitive, but with lower statistical accuracy

67

Discussion questions

1. What is the difference between similarity and homology? When does high identity not imply homology? What conclusions can be drawn from homology?
2. What is the difference between homology and common ancestry?
3. When the *M. janaschii* genome was first sequenced, Venter and his colleagues stated that almost 60% of the open reading frames (proteins or genes) were novel to this organism. (For eubacterial like *E. coli* or *H. influenzae*, a similar number would be 20 - 40%). On what would they base such a statement? Is it likely to be correct?
4. Name two reasons why protein sequence comparison is more effective (longer evolutionary look-back time) than DNA sequences?
5. What is the range of an expectation value ($E()$ -value)? If you compare a sequence to 50,000 random (unrelated) sequences, what should the expectation value for the highest of the 50,000 similarity scores be (on average)?
6. In a sequence similarity database search, you identify a statistically significant similarity ($E() < 0.005$), but the alignment is relatively short (50 aa). How might you determine whether the alignment reflects a genuine homology, or a random sequence match?
7. How can a sequence be homologous if you search a small database (e.g. human, 40,000 sequences), but not share significant similarity if you search a complete database (>4 million sequences)?
8. What scoring matrix should be used to identify protein orthologs that have diverged over the past 100 My (e.g. human/mouse)?
9. What scoring matrix should be used when comparing Illumina 90 nt reads against a protein database?

68

Algorithms in Bioinformatics

Jim Tisdall

[Programming for Biology](#)

Lecture Notes

1. [The Problem](#)
2. [Time and Space and Algorithms](#)
3. [Using Less Time](#)
4. [Using Less Space](#)
5. [Profiling](#)
6. [Parallel Processing](#)

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 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

$O(2^n)$.

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

Examples:

$O(2^n)$ computations: *intractable* (e.g. *exponential*) is *bad*

$O(n^2)$ 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 n^2 is 10,000 which is still not bad. However, 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 occurrence 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 occurrence 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 occurrence 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
AAAAAAAACGTACGTACGTCCGCGCGCGCGCACGTACGTACG
TGGGGGGGGGGGGGGCCCCCCCCCGGGGGGGGGGGAAAAAAACG
TACGTACGTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
```

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. Hmm. 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 *interconnected networks*) 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.

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
AAAAAAAACGTACGTACGTCCGCGCGCGCGCACGTACGTACG
TGGGGGGGGGGGGGGCCCCCCCCCGGGGGGGGGGGAAAAAAACG
TACGTACGTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
```

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