Some Programming Required

- Existing software tools can rarely do all steps
- Source data in different files may be formatted differently
- We need to bridge the differences with simple transformations
- Some programming ability is necessary for every project
Programming Languages

Two major groups

• Compiled languages → the output is (usually) a standalone program that can be run

• Interpreted languages → requires the presence of another software that in turn will run the software

• Common question: which language can solve a given problem?
TuringCompleteness

• All languages can be used to solve **any** computable problem.

• But the solutions may end up radically different!

• Tradeoffs between simplicity, speed of development and the speed of execution.
It is all about the libraries!

- Modern software development is all about reusing existing functionality

- Programming languages are turning into so-called “very high level programming environments”

- Most people rarely code full algorithms – they piece together existing functionality
Possibly the first scripting language: awk

• Appeared in 1977, strong UNIX (1972) roots

• It is the precursor of languages such as Perl (1987) and Python (1989)

• Has fallen into disuse for a quite a while, predicted to fail.

• Large text based datasets have lead to a resurgence of the language.
For a few years after the release of Perl in 1987, *awk* remained competitive simply because it had a smaller, faster implementation. But as the cost of compute cycles and memory dropped, the economic reasons for favoring a special-purpose language that was relatively thrifty with both lost their force. Programmers increasingly chose to do awklike things with Perl or (later) Python, rather than keep two different scripting languages in their heads.[90] By the year 2000 *awk* had become little more than a memory for most old-school Unix hackers, and not a particularly nostalgic one.

Not quite right: **The more things change the more they stay the same.**

There is now a resurgence of *awk* because the amount of data made our computers weak again.
The structure of an **awk** program

```awk ' pattern { action } '```

Line oriented

1. Tries to match the pattern to the line
2. If it is a match execute the action
3. Automatically splits the data by whitespace.
4. No pattern means match everything.
Awk **automatically splits** the input by whitespace (spaces and tabs) and assigns names to them:

- $0$ the entire line
- $1$ first field
- $2$ second field
- ...
- **NF** the number of fields
- **NR** the number of the current line
Operators

• \( + - \ast / \) for numerical context

• \( > < \) for comparison

• \( \% \) modulo division (remainder of division)

• \(<\text{space}>\) string concatenation

• \( ==, != \) equal, not equal

• \( \sim, !\sim \) match, no match (regular expressions)
AWK program in action

```bash
$ cat NC.gff | awk ' $3=="gene" { print $3, $5-$4 + 1 } ' | head -5
gene  2971
gene  1376
gene  1505
gene  2406
gene  1453
(env)
```

matching

action
You may also put the program into a file

```
{ print NR, NF, $3, $5 - $4 + 1 }
```

```
$ cat NC.gff | awk -f name.awk | head -5
1 1 1
2 9 feature 1
3 9 source 18959
4 9 5'UTR 55
5 9 gene 2971
```
List the gene names and sizes

Write one step at a time

```bash
#!/bin/bash

$3 == "gene" {
    split($9, x, ";")
    split(x[1], y, " ")
    print $3, y[2]
}
```

```
ialbert@grit ~/edu/lec17
$ cat NC.gff | awk -f split.awk | head -5
gene   "NP"
gene   "VP35"
gene   "VP40"
gene   "GP"
gene   "VP30"
(env)
ialbert@grit ~/edu/lec17
$ 
```
Full program

```bash
$3 == "gene" {  
    # Split the 9th column by ;
    split($9, x, ";\"");
    # The gene name is in the first resulting element. # Split that by space. The gene name is the second # element.
    split(x[1], y, " ");
    # Remove the double quotes around the gene name
    name = y[2]
    # Global substitution of " with empty space. # Since " is also a special character we have to # write it as \"
    gsub(""", "", name)

    # Print the type of the feature, the name
    print $3, name, $5 - $4 + 1
}
```

```
$ cat NC.gff | awk -f split.awk | head -5
gene NP 2971
gene VP35 1376
gene VP40 1505
gene GP 2406
gene VP30 1453
```
The whitespace curse: spaces and tabs

- Many tools will auto-split by whitespace → this was thought to be convenient but is also the source of extremely subtle errors → leads to a column shift in a tab file if a field contains spaces

- **Always specify the character to be split by!**

- This refers to programming languages as well!

  Do not use the `split()` methods with their default behavior (Python, Perl etc) unless you perfectly understand what they do)
Customize awk to use tabs as both the input and output field separator

```
alias awk="awk -F '\t' -v OFS='\t'"
```

Tip: you can add this to the `.profile` or `.bashrc` file in your root folder so that it is activate all the time

**Note:** file names that start with a dot . are only listed if you do a `ls -a`

```
ialbert@porthos ~/work
$ tail -5 ~/.profile
alias ls='ls -hG'
alias mv='mv -i'
alias rm='rm -i'
alias awk="awk -F '\t' -v OFS='\t'"

ialbert@porthos ~/work
$`
```
Special patterns

- **BEGIN** → before the stream starts
- **END** → after the stream ends

```bash
# a running tally of the total length of genomic intervals in the data

BEGIN { num = 0 }

{ size = $5 - $4 + 1; num += size; }

END { print num }
```

```bash
ialbert@porthos ~/work
$ cat short.gff | awk -f counter.awk | more
8886525
ialbert@porthos ~/work
$ 
```
Advanced awk – usually not needed

- conditionals: `if`
- loops: `for`, `while`
- `break`, `continue`
- associative data structures (hash, dictionary)

You can do all that though at that point it is probably better to learn Python

But you can do a lot with just basic awk!

Awk’s power comes from its simplicity – more complex programming is done with specialized programming languages
A few awk resources

Large number of resources, the quirky name makes it very searchable → **How to do X with awk?**

• How to use awk – see course webpage for links
Bioinformatics Survival Kit

Everyday bioinformatics functions

• Bioawk (**awk** with bioinformatics super powers)
• Seqtk (sequence toolkit)
• Tabix (indexed, searchable tab files)
• Tabtk (tab tookit)
Bioawk functions

When used with a format it will populate internal variables such as: $name, $seq, $start, $end etc.

It also works with both gzipped or regular text files: it’s magic no less
Bioawk examples

```
$ cat r1.fq | bioawk -c fastx '{ print $name }' | head -1
NC_002549_8570_9028_8:0:0_6:0:0_0/1
$
$ cat r1.fq | bioawk -c fastx '{ print $seq }' | head -1
AGGACCACCCCTGATCGACGACGATGATCATCCATAGATAATTATCGAGGTGCATAGCGTCAATCAAGGAG
$
$ cat mutations.gff | bioawk -c gff '{ print $start }' | head -1
244
$  ```
Using the alignments used in lecture 16 write awk scripts that perform the following:

- Print out those genomic indices that have a coverage that is over a half of the largest coverage observed in your data. What percent of your genome is that?

- Print the alignments for which the DNA fragment length is over half of the maximum observed DNA fragment length. (SAM column 9)

Hints:

- The simple solution is to break each problem into two steps as, find the value, filter with the value)
- Only show the first few lines (don’t submit all the data)