

# Intro to Annotation

Michael Schatz

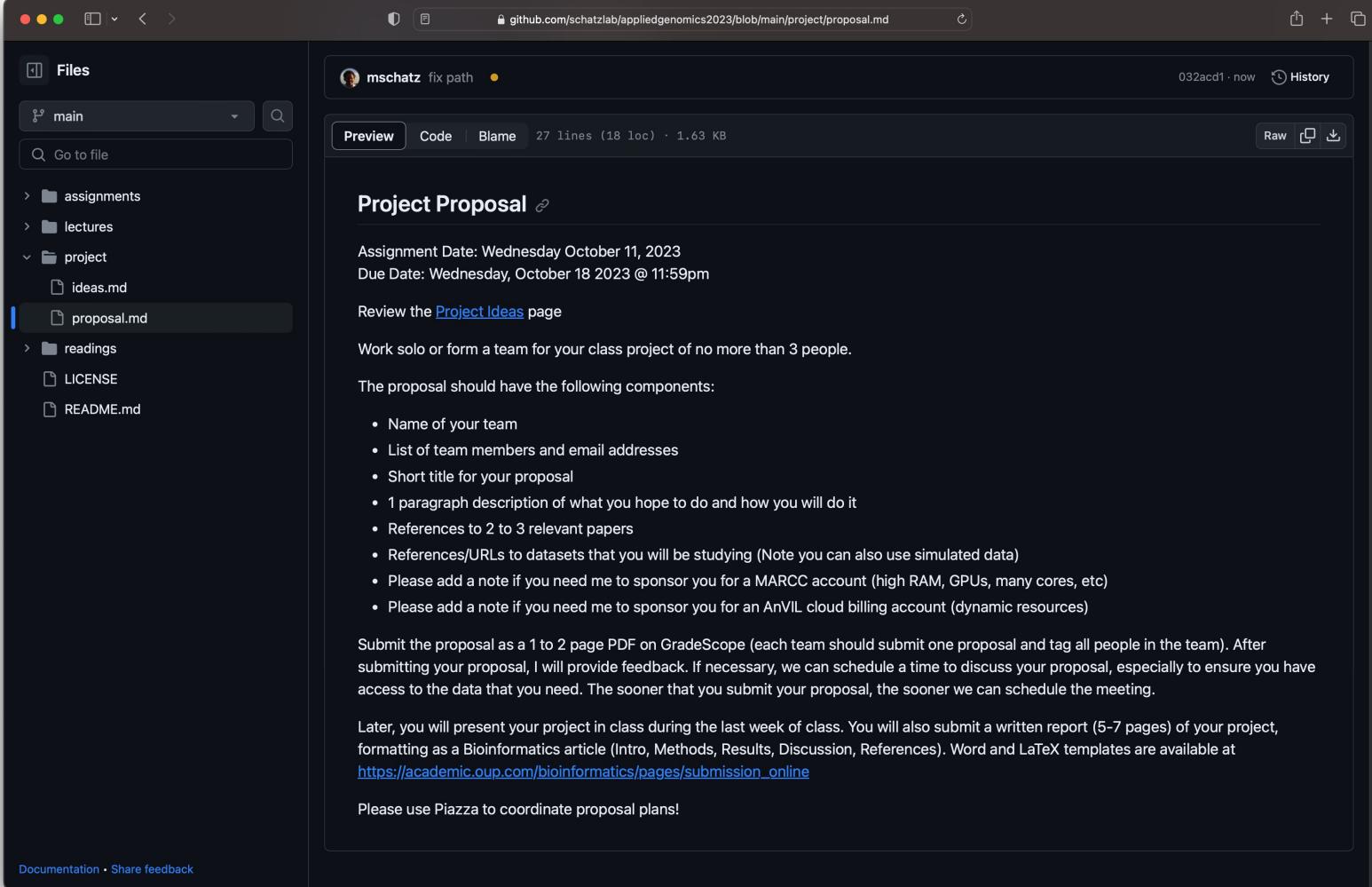
October 16, 2023

Lecture 14. Applied Comparative Genomics



# Project Proposal

## Due Wednesday Oct 18 by 11:59pm



The screenshot shows a GitHub repository page for the 'appliedgenomics2023' repository. The 'proposal.md' file is open in the preview tab. The file content is as follows:

## Project Proposal

Assignment Date: Wednesday October 11, 2023  
Due Date: Wednesday, October 18 2023 @ 11:59pm

Review the [Project Ideas](#) page

Work solo or form a team for your class project of no more than 3 people.

The proposal should have the following components:

- Name of your team
- List of team members and email addresses
- Short title for your proposal
- 1 paragraph description of what you hope to do and how you will do it
- References to 2 to 3 relevant papers
- References/URLs to datasets that you will be studying (Note you can also use simulated data)
- Please add a note if you need me to sponsor you for a MARCC account (high RAM, GPUs, many cores, etc)
- Please add a note if you need me to sponsor you for an AnVIL cloud billing account (dynamic resources)

Submit the proposal as a 1 to 2 page PDF on GradeScope (each team should submit one proposal and tag all people in the team). After submitting your proposal, I will provide feedback. If necessary, we can schedule a time to discuss your proposal, especially to ensure you have access to the data that you need. The sooner that you submit your proposal, the sooner we can schedule the meeting.

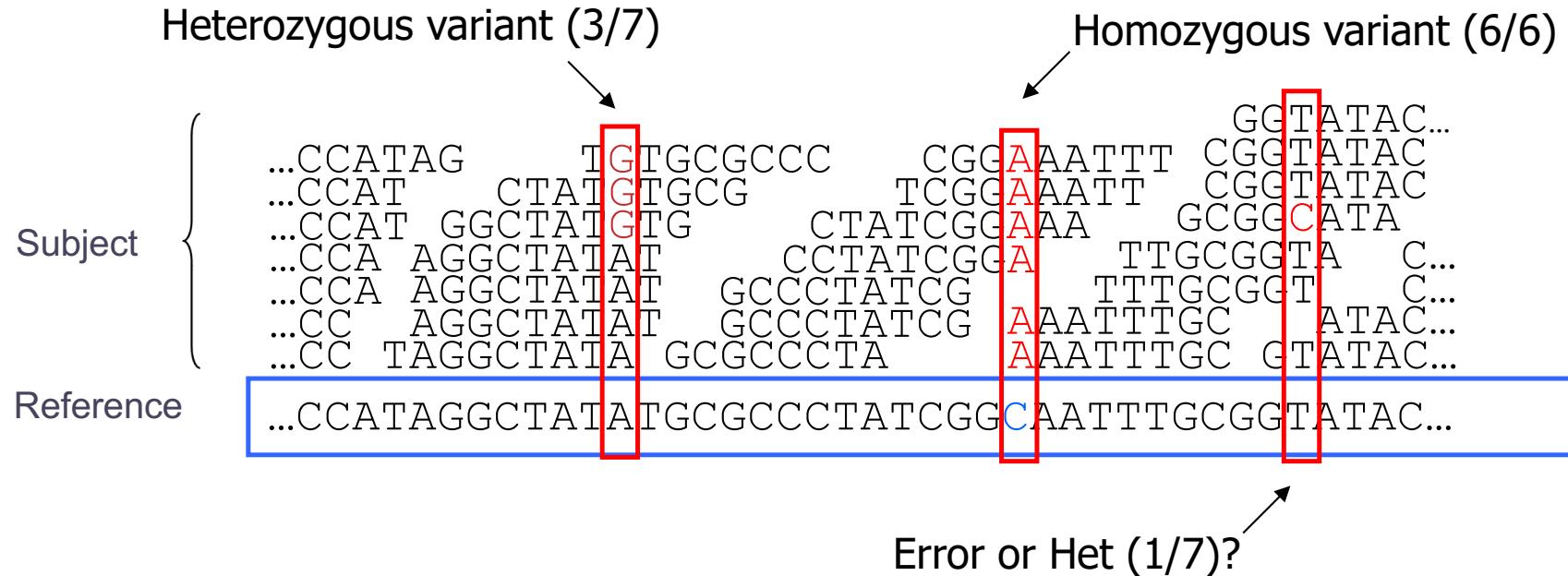
Later, you will present your project in class during the last week of class. You will also submit a written report (5-7 pages) of your project, formatting as a Bioinformatics article (Intro, Methods, Results, Discussion, References). Word and LaTeX templates are available at [https://academic.oup.com/bioinformatics/pages/submission\\_online](https://academic.oup.com/bioinformatics/pages/submission_online)

Please use Piazza to coordinate proposal plans!

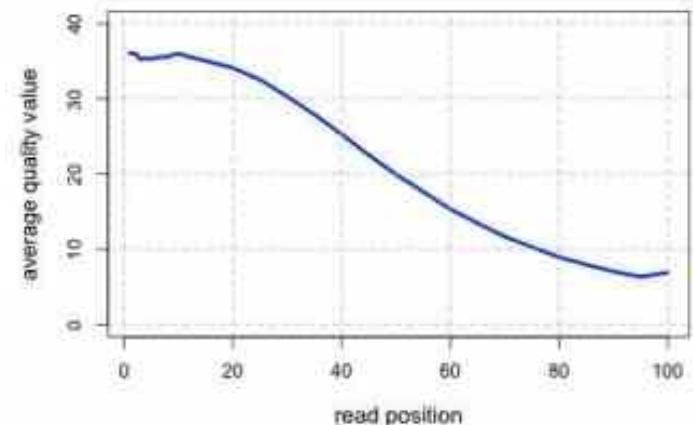
<https://github.com/schatzlab/appliedgenomics2023/blob/main/project/proposal.md>

Check Piazza for questions!

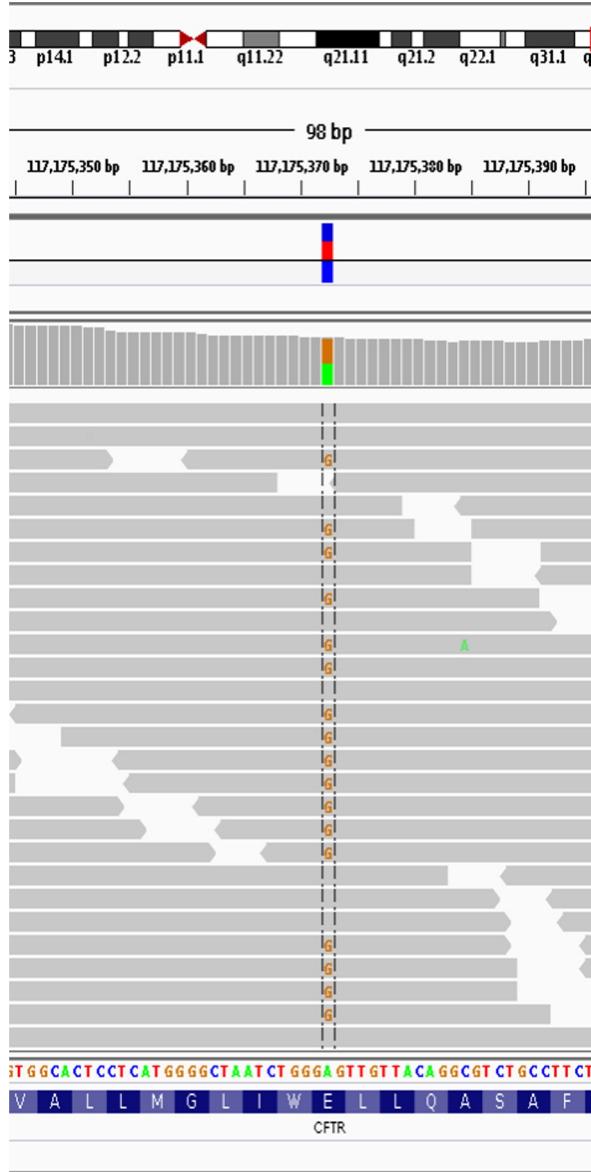
# Genotyping Theory



- If there were no sequencing errors, identifying SNPs would be very easy: any time a read disagrees with the reference, it must be a variant!
- Sequencing instruments make mistakes
  - Quality of read decreases over the read length
- A single read differing from the reference is probably just an error, but it becomes more likely to be real as we see it multiple times

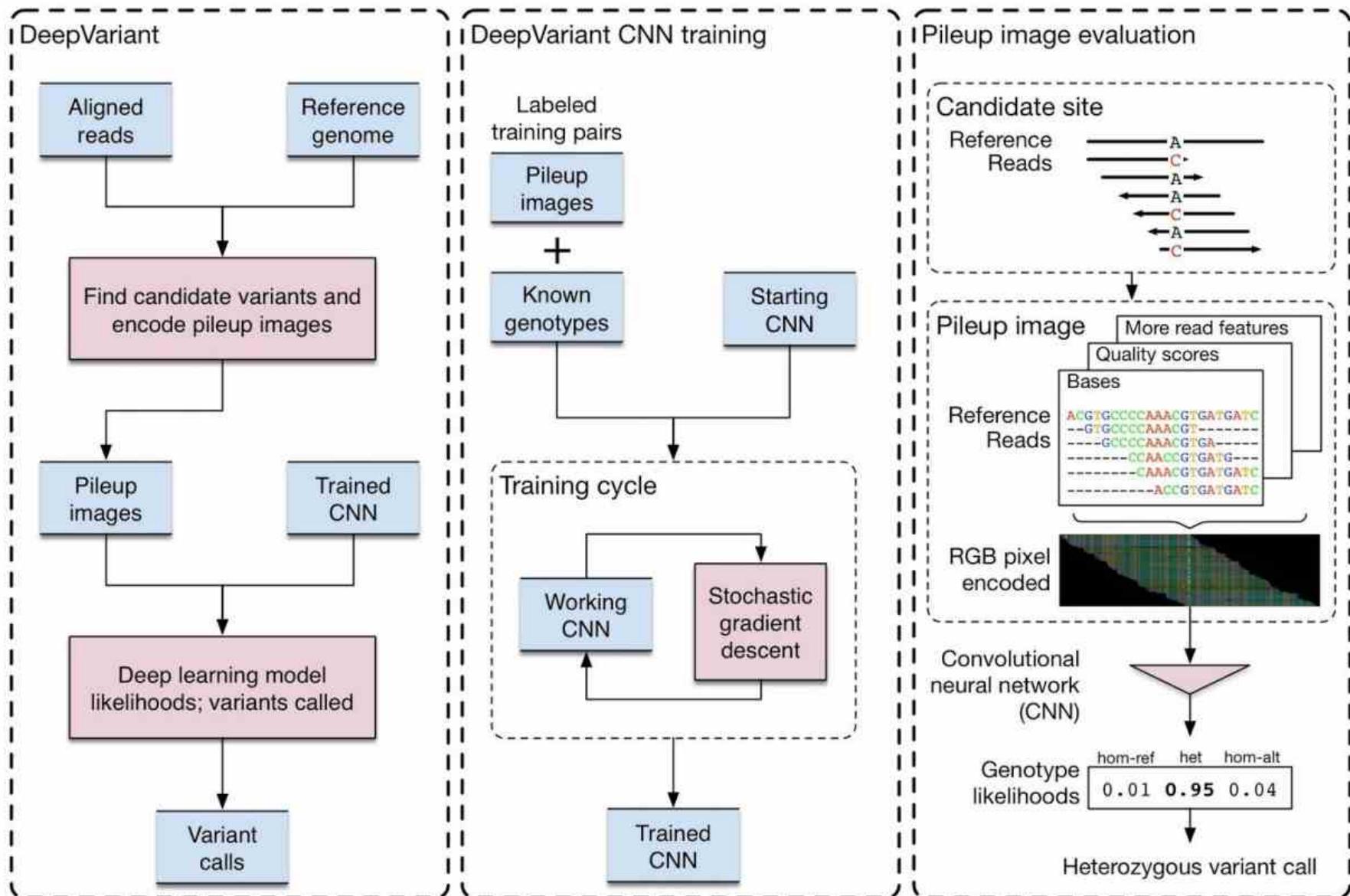


# What information is needed to decide if a variant exists?



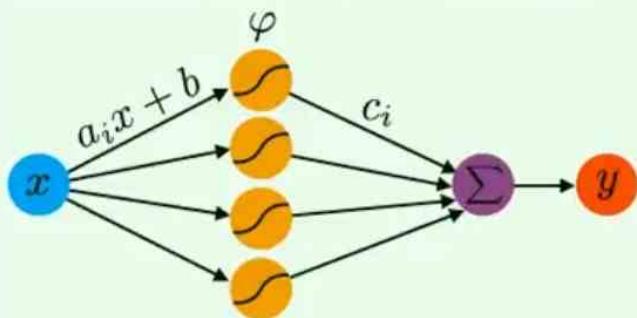
- Depth of coverage at the locus
- Bases observed at the locus
- The base qualities of each allele
- The strand composition
- Mapping qualities
- Proper pairs?
- Expected polymorphism rate

# DeepVariant



**Creating a universal SNP and small indel variant caller with deep neural networks**

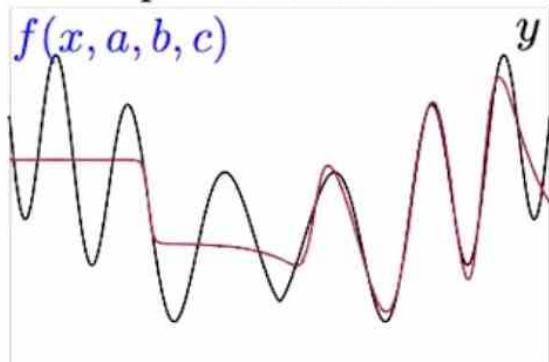
Poplin et al. (2018) Nature Biotechnology. <https://www.nature.com/articles/nbt.4235>



1 hidden layer perceptron:

$$y \approx f(x, a, b, c) \stackrel{\text{def.}}{=} \sum_{i=1}^p c_i \varphi(a_i x + b_i)$$

$p = 6$  neurons



$p = 20$  neurons



## Approximation by Superpositions of a Sigmoidal Function\*

G. Cybenko†

**Abstract.** In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of  $n$  real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

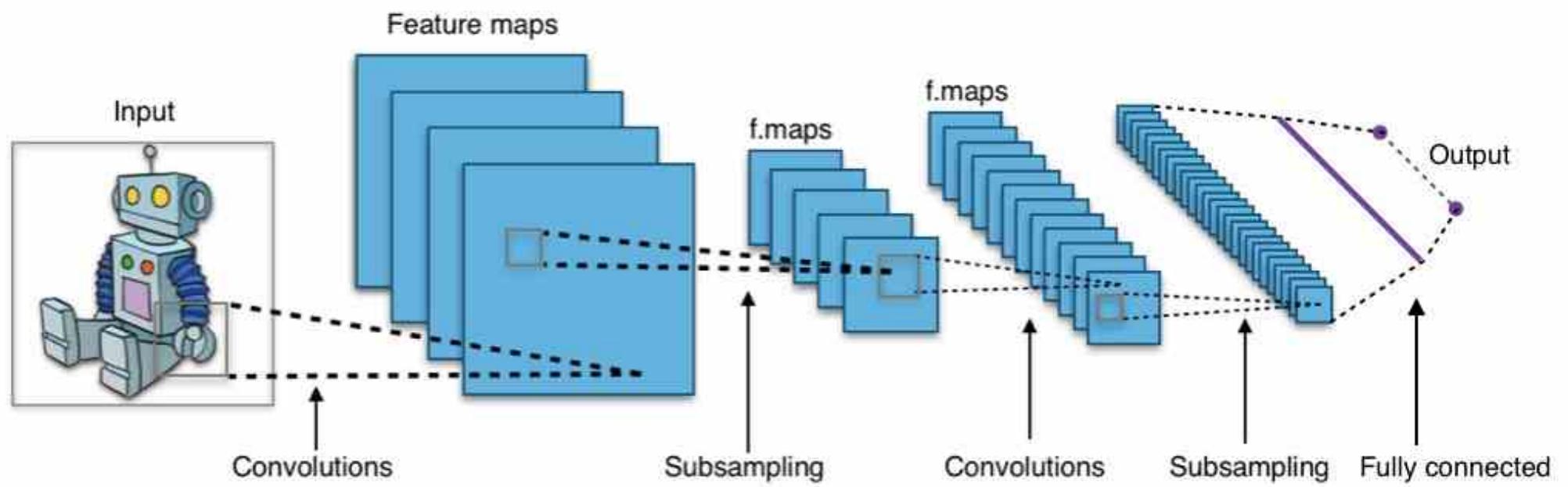
**Key words.** Neural networks, Approximation, Completeness.

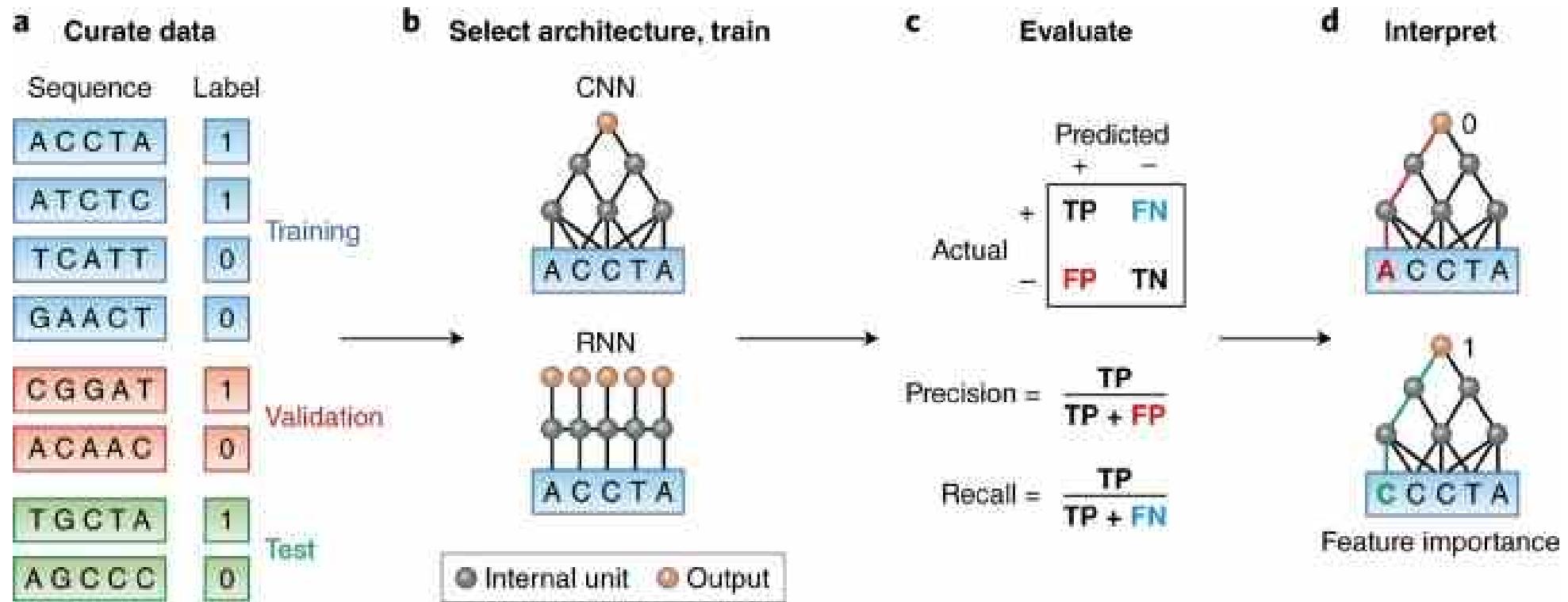


George Cybenko

## Approximation by superpositions of a sigmoidal function

Cybenko, G. (1989) Mathematics of Control Signal Systems doi: 10.1007/BF02551274





## A primer on deep learning in genomics

Zou et al (2019) Nature Genetics. <https://www.nature.com/articles/s41588-018-0295-5>

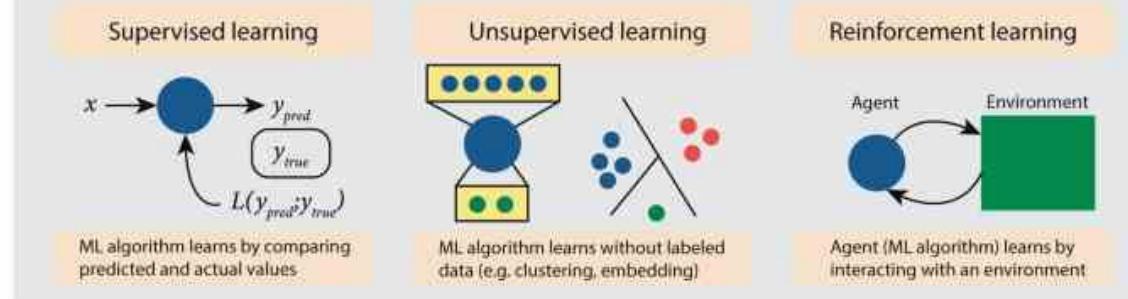
## Deep learning: new computational modelling techniques for genomics

Eraslan et al (2019) Nature Reviews Genetics. <https://www.nature.com/articles/s41576-019-0122-6>

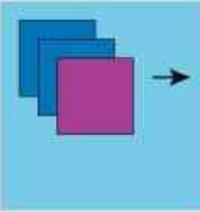
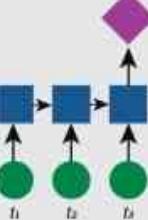
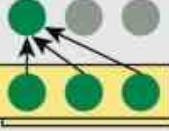
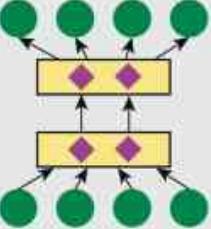
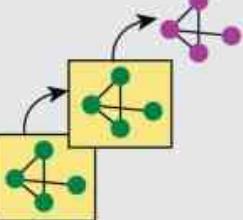
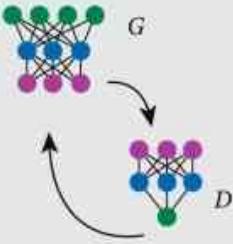
## Current progress and open challenges for applying deep learning across the biosciences

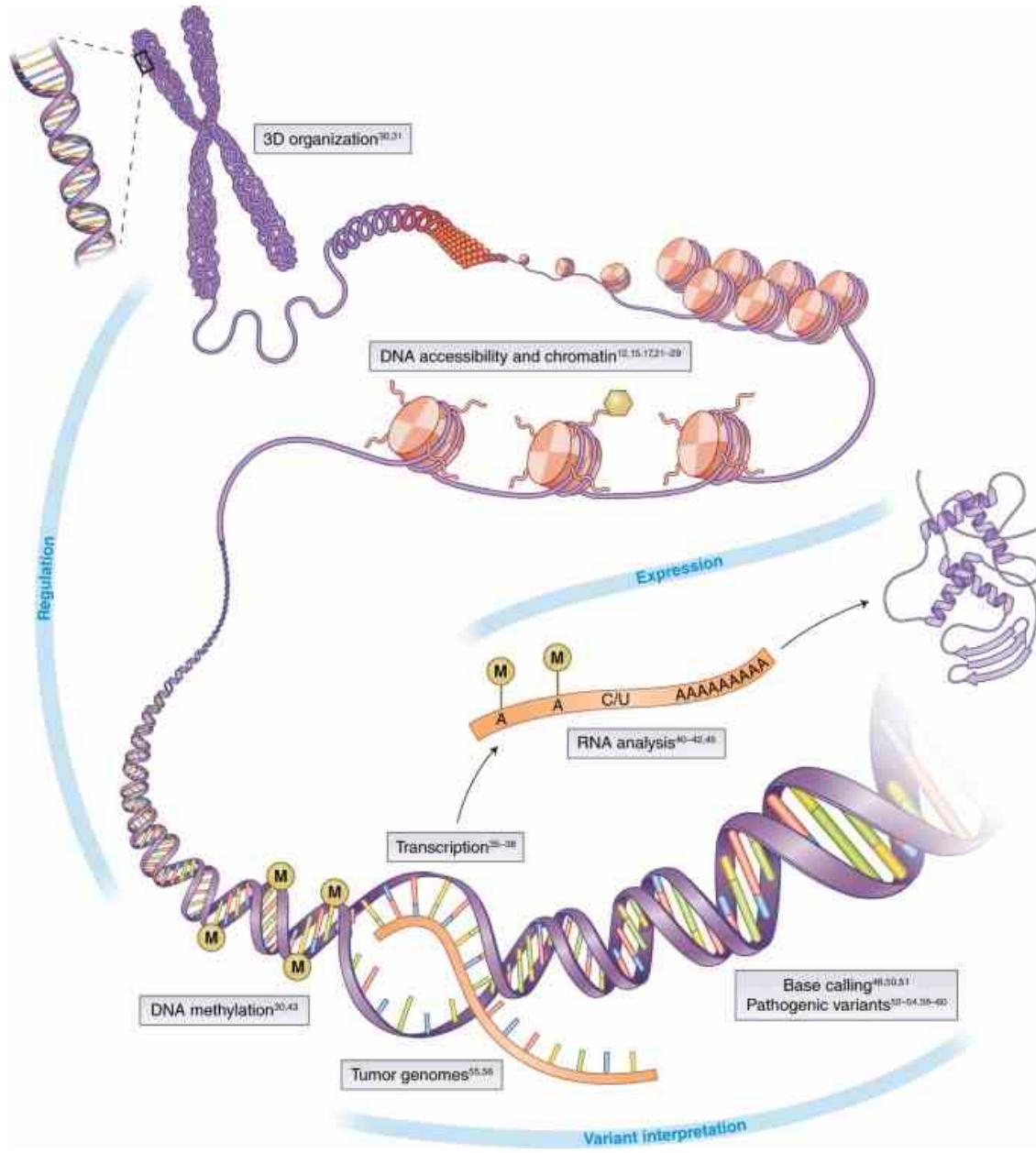
Sapoval et al (2022) Nature Communication. <https://www.nature.com/articles/s41467-022-29268-7>

## Machine learning scenarios



## Key deep learning architectures

<b>Convolutional NN (CNN)</b> Goal: Perform inference on data with local features Key idea: Learn shift-invariant filters 	<b>Recurrent NN (RNN)</b> Goal: Perform inference on temporal data Key idea: Learn temporal correlations via recurrent structure 	<b>Transformer</b> Goal: Perform inference on sequential data Key idea: Learn context based correlations via attention mechanism 	<b>Autoencoder (AE)</b> Goal: Embed high-dimensional data Key idea: Learn low-dimensional embedding of data 
<b>Graph NN (GNN)</b> Goal: Capture graph based dependencies in the data Key idea: Perform message passing between nodes in a layer 	<b>Generative Adversarial Network (GAN)</b> Goal: Generate samples from data distribution Key idea: Simultaneously train generator and discriminator 	<b>Denoising autoencoders (DAE)</b> are autoencoder models that learn low dimensional embeddings of noisy high dimensional data, i.e. inputs that differ by a small amount of noise give rise to a similar embedding vector.  <b>Attention mechanism</b> mimics cognitive attention by learning importance weights for the inputs based on the whole input context (e.g. in a task of translating codons to amino acids attention mechanism will learn to give higher weight to the first two nucleic acids). Attention is the key part of transformer models, but can also be applied in conjunction with other layer types.  <b>Convolutional layers</b> have dimension which indicates the dimension of learned filters. Thus, we can have a 1-dimensional convolutional layer for sequences, 2-dimensional layer for matrices, and so on.  <b>Graph convolutional network (GCN)</b> is a graph neural network with convolutional layers defined by the topology of the graph. Thus instead of passing neighboring sequence or matrix entries through a filter, graph defined neighborhoods are used.	



A primer on deep learning in genomics

Zou et al (2019) Nature Genetics. <https://www.nature.com/articles/s41588-018-0295-5>

# Annotation

# Goal: Genome Annotations

aatgcatgcggctatgcta atgc atgcggctatgcta agc tggatccgat gaca atgc atgcggctatgcta at  
gcatgcggctatgcaagctggatccgatgactatgcta agc tggatccgat gaca atgc atgcggctatgct  
aatgaatggtcttggattac ttgaa atgcta agc tggatccgat gaca atgc atgcggctatgcta atgaa  
tgg tcttggattac ttgaa atgcta atgc atgcggctatgcta agc tggatccgat gaca atgc atgc  
gctatgcta atgc atgcggctatgca agc tggatccgatgactatgcta agc tggatccgat gcta atgc  
gctatgcta agc tggatccgat gaca atgc atgcggctatgcta atgc atgcggctatgca agc tggatcc  
gcggctatgcta atgaa atgg tcttggattac ttgaa atgcta agc tggatccgat gaca atgc atgcggct  
atgcta atgaa atgg tcttggattac ttgaa atgcta atgc atgcggctatgcta agc tggatgc  
gctatgcta agc tggatccgat gaca atgc atgcggctatgcta atgc atgcggctatgca agc tggatcc  
atgactatgcta agc tgc ggctatgcta atgc atgcggctatgcta agc tgc ggctatgcta agc tggat  
gcatgcggctatgcta agc tggatccgat gaca atgc atgcggctatgcta atgc atgcggctatgca agc  
ggatccgatgactatgcta agc tgc ggctatgcta atgc atgcggctatgcta agc tgc ggctatgcta  
gtcttggattac ttgaa atgcta agc tggatccgat gaca atgc atgcggctatgcta atgaa atgg tcttgg  
attac ttgaa atgcta atgc atgcggctatgcta agc tggatgc atgcggctatgcta agc tggatcc  
cgat gaca atgc atgcggctatgcta atgc atgcggctatgca agc tggatccgatgactatgcta agc tgc  
gctatgcta atgc atgcggctatgcta agc tgc

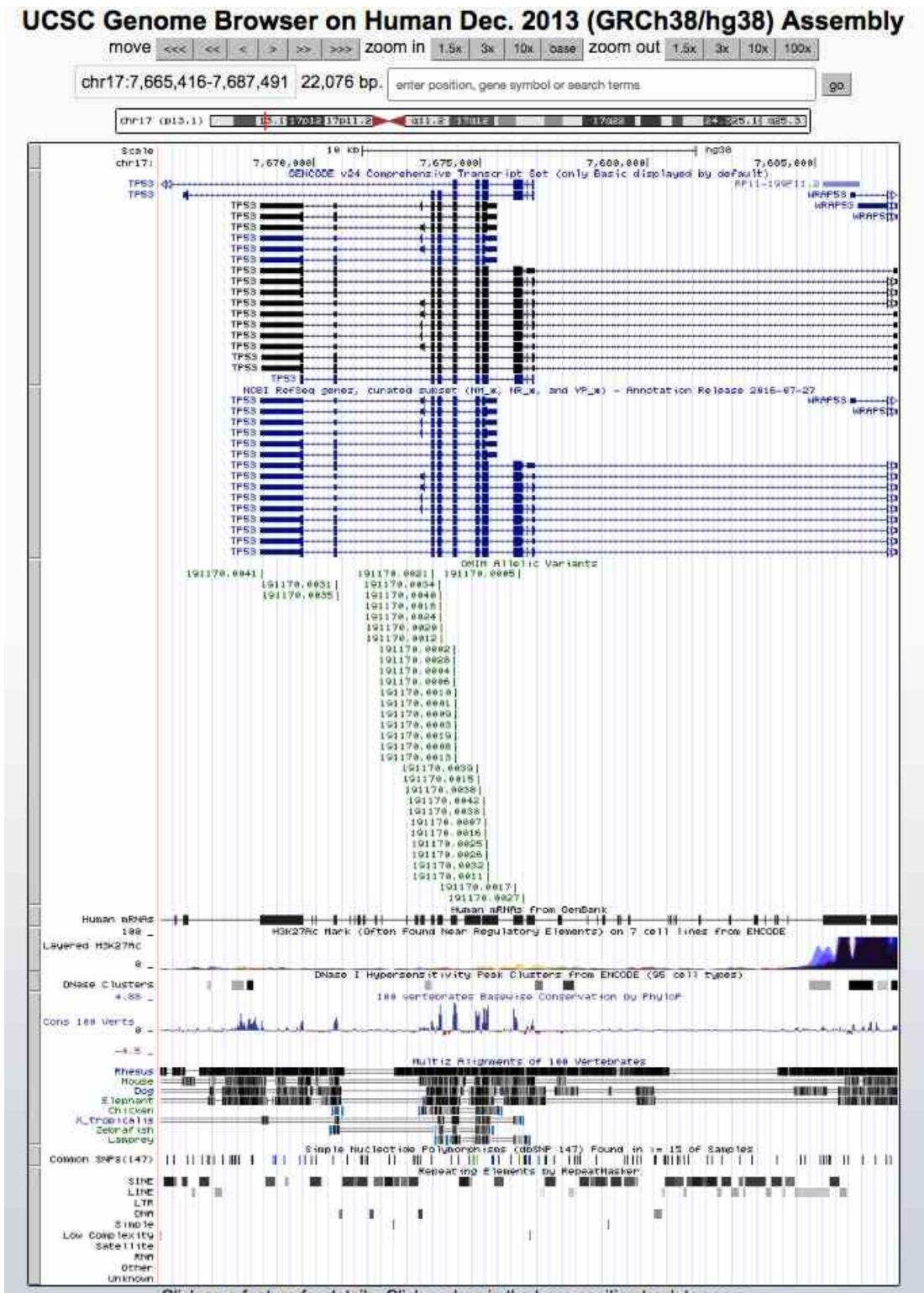
# Goal: Genome Annotations

aatgcatgcggctatgctaattgcattgcggctatgctaagctggatccgatgacaatgcattgcggctatgctaatt  
gcatgcggctatgcaaggctggatccgatgactatgctaagctggatccgatgacaatgcattgcggctatgctaatt  
aatgaatggtcttggattttaccttggaaatgtctaattgcattgcggctatgctaagctggatccgatgacaatgcattgcggctatgctaatt  
tggtcttggattttaccttggaaatgtctaattgcattgcggctatgctaagctggatccgatgacaatgcattgcg  
gctatgctaattgcattgcggctatgcaaggctggatccgatgactatgctaagctgcggctatgctaattgcattgcg  
gctatgctaagctggatccgatgacaatgcattgcggctatgctaattgcattgcggctatgctaagctggatcc  
gctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
atgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
gctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
atgactatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
gcatgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcg  
ggatccgatgactatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcg  
gtcttggattttaccttggaaatgtctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
gatttaccttggaaatgtctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
cgatgacaatgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
gctatgctaattgcattgcggctatgctaattgcattgcg

Gene!

# What are genome intervals?

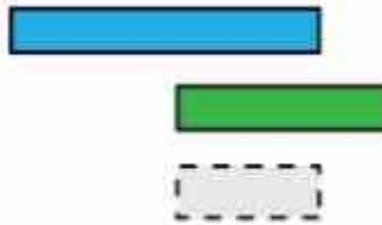
- Genetic variation:
  - SNPs: 1bp
  - Indels: 1-50bp
  - SVs: >50bp
- Genes:
  - exons, introns, UTRs, promoters
- Conservation
- Transposons
- Origins of replication
- TF binding sites
- CpG islands
- Segmental duplications
- Sequence alignments
- Chromatin annotations
- Gene expression data
- ...
- **Your own observations and data: put them into context!**



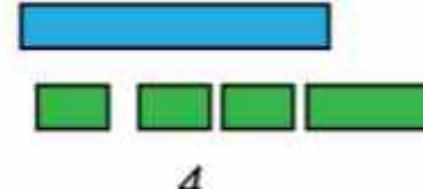
# BEDTools to the rescue!



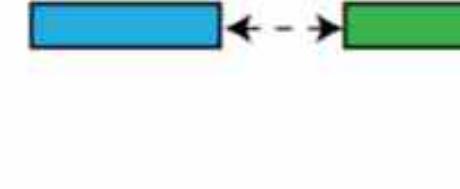
*Intersect*



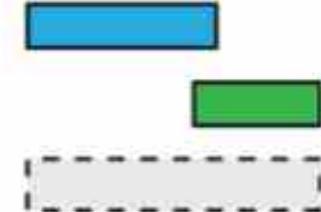
*Count overlaps*



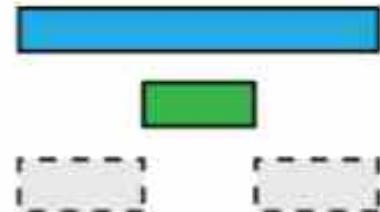
*Distance*



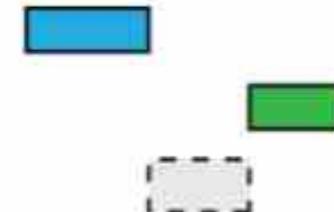
*Merge*



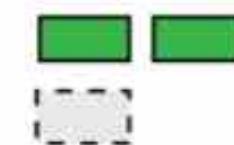
*Subtract overlaps*



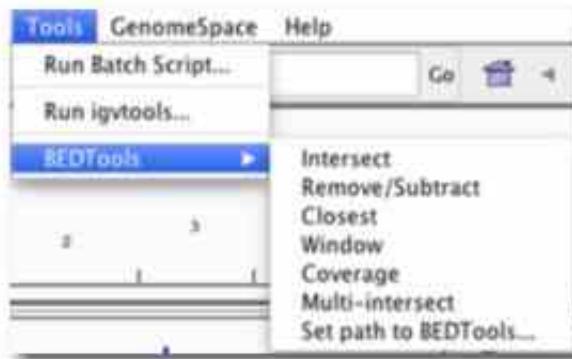
*Complement*



*Closest*



# Getting & Using BEDTools



Integrated into IGV

## BEDTools

- [Intersect BAM alignments with intervals in another file](#)
- [Count intervals in one file overlapping intervals in another file](#)
- [Create a histogram of genome coverage](#)
- [Create a BedGraph of genome coverage](#)
- [Convert from BAM to BED](#)
- [Merge BedGraph files](#)
- [Intersect multiple sorted BED files](#)

In Galaxy Toolshed

The screenshot shows the homepage of the bedtools documentation at bedtools.readthedocs.io/en/latest/. The page features a large red logo with a white 'b' inside a shield shape, followed by the text 'bedtools'. Below the logo, a paragraph describes bedtools as a powerful toolset for genome arithmetic, mentioning its use for tasks like intersect, merge, count, complement, and shuffle genomic intervals. It also notes its development in the Quinlan laboratory at the University of Utah.

**bedtools: a powerful toolset for genome arithmetic**

Collectively, the **bedtools** utilities are a swiss-army knife of tools for a wide-range of genomics analysis tasks. The most widely-used tools enable *genome arithmetic*: that is, set theory on the genome. For example, **bedtools** allows one to *intersect*, *merge*, *count*, *complement*, and *shuffle* genomic intervals from multiple files in widely-used genomic file formats such as BAM, BED, GFF/GTF, VCF. While each individual tool is designed to do a relatively simple task (e.g., *intersect* two interval files), quite sophisticated analyses can be conducted by combining multiple bedtools operations on the UNIX command line.

**bedtools** is developed in the Quinlan laboratory at the University of Utah and benefits from fantastic contributions made by scientists worldwide.

## Tutorial

We have developed a fairly comprehensive [tutorial](#) that demonstrates both the basics, as well as some more advanced examples of how bedtools can help you in your research. Please have a look.

## Interesting Usage Examples

In addition, here are a few examples of how bedtools has been used for genome research. If you have interesting examples, please send them our way and we will add them to the list.

- Coverage analysis for targeted DNA capture. Thanks to [Stephen Turner](#).
- Measuring similarity of DNase hypersensitivity among many cell types
- Extracting promoter sequences from a genome
- Comparing intersections among many genome interval files
- RNA-seq coverage analysis. Thanks to [Erik Minikel](#).
- Identifying targeted regions that lack coverage. Thanks to [Brent Pedersen](#).
- Calculating GC content for CCDS exons.
- Making a master table of ChromHMM tracks for multiple cell types.

## Table of contents

Extensive Documentation and Examples

# BED Format

***BED (Browser Extensible Data) format provides a flexible way to define intervals.***

***The first three required BED fields are:***

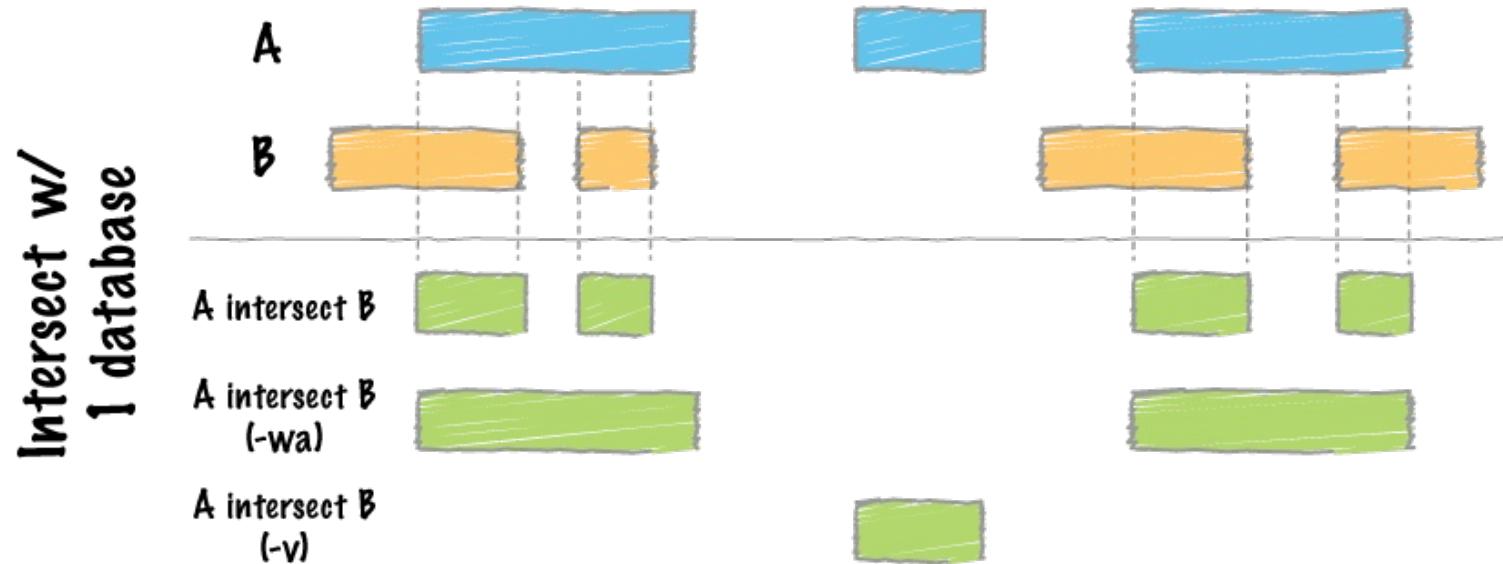
1. chrom The name of the chromosome (e.g. chr3, chrY, chr2\_random) or scaffold (e.g. scaffold10671).
2. chromStart The starting position of the feature in the chromosome or scaffold. The first base in a sequence is numbered 0.
3. chromEnd The ending position of the feature in the chromosome or scaffold.  
The chromEnd base is not included in the display of the feature. For example, the first 100 bases of a chromosome are defined as chromStart=0, chromEnd=100, and span the bases numbered 0-99.

***The 9 additional optional BED fields are:***

1. name - Defines the name of the BED line
2. score - A score between 0 and 1000
3. strand - Defines the strand. Either "." (=no strand) or "+" or "-".
4. thickStart - The starting position at which the feature is drawn thickly
5. thickEnd - The ending position at which the feature is drawn thickly (for example the stop codon in gene displays).
6. itemRgb - An RGB value of the form R,G,B (e.g. 255,0,0).
7. blockCount - The number of blocks (exons) in the BED line.
8. blockSizes - A comma-separated list of the block sizes. The number of items in this list should correspond to blockCount.
9. blockStarts - A comma-separated list of block starts. All of the blockStart positions should be calculated relative to chromStart. The number of items in this list should correspond to blockCount.

```
## genes.bed has: chrom, txStart, txEnd, name, num_exons, and strand
$ head -n4 genes.bed
chr1    134212701    134230065    Nuak2      8      +
chr1    134212701    134230065    Nuak2      7      +
chr1    33510655     33726603     Prim2,     14     -
chr1    25124320     25886552     Bai3,     31     -
```

# BEDTools Intersect



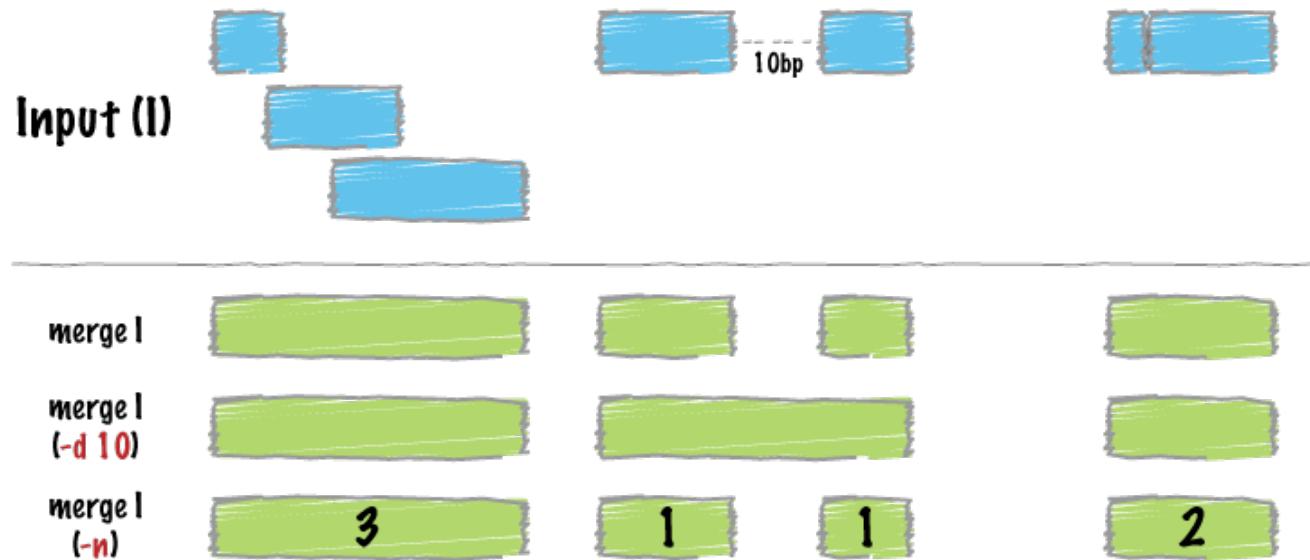
**What exons are hit by SVs?**

```
$ cat A.bed  
chr1 10 20  
chr1 30 40  
  
$ cat B.bed  
chr1 15 20  
  
$ bedtools intersect -a A.bed -b B.bed -wa  
chr1 10 20
```

**What parts of exons are hit by SVs?**

```
$ cat A.bed  
chr1 10 20  
chr1 30 40  
  
$ cat B.bed  
chr1 15 20  
  
$ bedtools intersect -a A.bed -b B.bed  
chr1 15 20
```

# BEDTools Merge



**What parts of the genome are exonic?**

```
bedtools merge -i exons.bed | head -n 20
chr1    11873   12227
chr1    12612   12721
chr1    13220   14829
chr1    14969   15038
chr1    15795   15947
chr1    16606   16765
chr1    16857   17055
chr1    17222   17260
```

**Note input must be sorted!**

```
sort -k1,1 -k2,2n foo.bed > foo.sort.bed
```

# BEDTools commands

annotate	getfasta	overlap
bamtobed	groupby	pairtobed
bamtofastq	groupby	pairstopair
bed12tobed6	igv	random
bedpetobam	intersect	reldist
bedtobam	jaccard	shift
closest	links	shuffle
cluster	makewindows	slop
complement	map	sort
coverage	maskfasta	subtract
expand	merge	tag
flank	multicov	unionbedg
fisher	multiinter	window
genomcov	nuc	

<http://bedtools.readthedocs.io/en/latest/content/bedtools-suite.html>

# Plane Sweep

pos:	1	5	10	15	20	25	30	35	40	45	50	55
pos:												
r1:	[=====]											
r2:	[=====]											
r3:	[=====]											
r4:	[=====]											
r5:	[=====]											

## ***The basic algorithm works like this:***

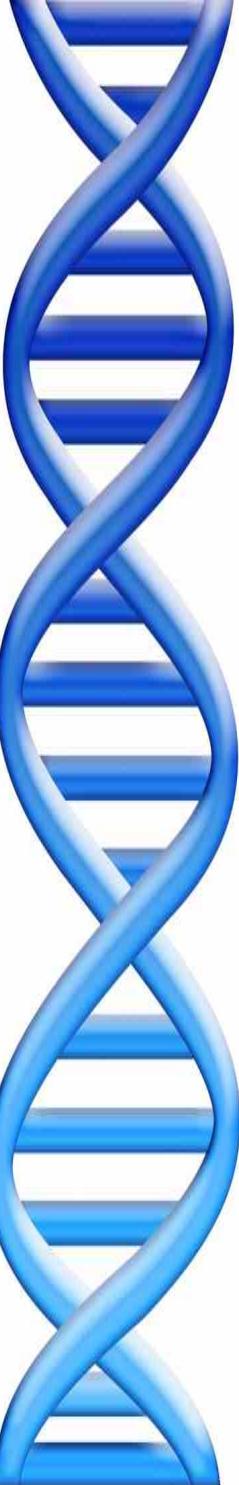
- Assume layout is in sorted order by start position (or explicitly sort by start position)
- use a “list” to track how many reads currently intersect the plane keyed by end coord
  - the number of elements in the list corresponds to the current depth
- walking from start position to start position
  - check to see if we past any read ends
  - coverage goes down by one when a read ends
  - coverage goes up by one when new read is encountered

See lecture notes!

# Goal: Genome Annotations

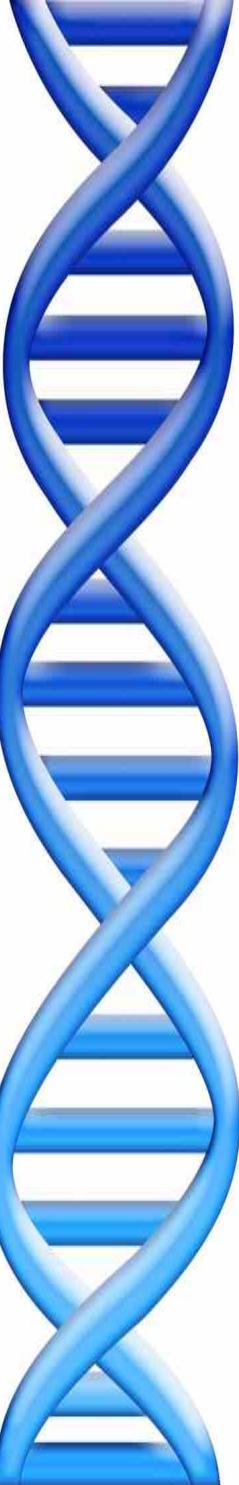
aatgcatgcggctatgctaattgcattgcggctatgctaagctggatccgatgacaatgcattgcggctatgctaatt  
gcatgcggctatgcaaggctggatccgatgactatgctaagctggatccgatgacaatgcattgcggctatgctaatt  
aatgaatggtcttggattttaccttggaaatgtctaattgcattgcggctatgctaagctggatccgatgacaatgcattgcggctatgctaatt  
tggtcttggattttaccttggaaatgtctaattgcattgcggctatgctaagctggatccgatgacaatgcattgcg  
gctatgctaattgcattgcggctatgcaaggctggatccgatgactatgctaagctgcggctatgctaattgcattgcg  
gctatgctaagctggatccgatgacaatgcattgcggctatgctaattgcattgcggctatgctaagctggatcc  
gctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
atgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
gctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
atgactatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
gcatgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcg  
ggatccgatgactatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcggctatgctaattgcg  
gtcttggattttaccttggaaatgtctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
gatttaccttggaaatgtctaattgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
cgatgacaatgcattgcggctatgctaattgcattgcggctatgctaattgcattgcg  
gctatgctaattgcattgcggctatgctaattgcattgcg

Gene!



# Outline

1. Alignment to other genomes
2. Prediction aka “Gene Finding”
3. Experimental & Functional Assays



# Outline

1. Alignment to other genomes
2. Prediction aka “Gene Finding”
3. Experimental & Functional Assays

# Basic Local Alignment Search Tool

- Rapidly compare a sequence  $Q$  to a database to find all sequences in the database with a score above some cutoff  $S$ .
  - Which protein is most similar to a newly sequenced one?
  - Where does this sequence of DNA originate?
- Speed achieved by using a procedure that typically finds “most” matches with scores  $> S$ .
  - Tradeoff between sensitivity and specificity/speed
    - Sensitivity – ability to find all related sequences
    - Specificity – ability to reject unrelated sequences

# Seed and Extend

FAKDFLAGGVAAAISKTAVAPIERVKLLLQVQHASKQITADKQYKGIIDCVVRIPKEQGV

FLIDLASGGTAAAVSKTAVAPIERVKLLLQVQDASKAIAVDKRYKGIMDVLIRVPKEQGV

- Homologous sequences are likely to contain a **short high scoring word pair**, a seed.
  - Smaller seed sizes make the sense more sensitive, but also (much) slower
  - Typically do a fast search for prototypes, but then most sensitive for final result
- BLAST then tries to extend high scoring word pairs to compute **high scoring segment pairs** (HSPs).
  - Significance of the alignment reported via an e-value

# Seed and Extend

```
FAKDFLAGGVAAAISKTAVAPIERVKLLLQVQHASKQITADKQYKGIIDCVVRIPKEQGV  
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  
FLIDLASGGTAAAVSKTAVAPIERVKLLLQVQDASKAIAVDKRYKGIMDVLIRVPKEQGV
```

- Homologous sequences are likely to contain a **short high scoring word pair**, a seed.
  - Smaller seed sizes make the sense more sensitive, but also (much) slower
  - Typically do a fast search for prototypes, but then most sensitive for final result
- BLAST then tries to extend high scoring word pairs to compute **high scoring segment pairs** (HSPs).
  - Significance of the alignment reported via an e-value

# BLAST E-values

E-value = the number of HSPs having alignment score S (or higher) expected to occur by chance.

- Smaller E-value, more significant in statistics
- Bigger E-value, less significant
- Over 1 means expect this totally by chance  
(not significant at all!)

The expected number of HSPs with the score at least S is :

$$E = K * n * m * e^{-\lambda S}$$

K,  $\lambda$  are constant depending on model

n, m are the length of query and sequence

E-values quickly drop off for better alignment bits scores

# Very Similar Sequences

Query: HBA\_HUMAN Hemoglobin alpha subunit

Sbjct: HBB\_HUMAN Hemoglobin beta subunit

Score = 114 bits (285), Expect = 1e-26

Identities = 61/145 (42%), Positives = 86/145 (59%), Gaps = 8/145 (5%)

Query 2 LSPADKTNVAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF-----DLSHGSAQV 55  
L+P +K+ V A WGKV + E G EAL R+ + +P T+ +F F D G+ +V

Sbjct 3 LTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKV 60

Query 56 KGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPA 115  
K HGKKV A ++ +AH+D++ + LS+LH KL VDP NF+LL + L+ LA H

Sbjct 61 KAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFGK 120

Query 116 EFTP AVHASLDKFLASVSTVLTSKY 140  
EFTP V A+ K +A V+ L KY

Sbjct 121 EFTPPVQAAYQKVVAGVANALAHKY 145

# Quite Similar Sequences

Query: HBA\_HUMAN Hemoglobin alpha subunit

Sbjct: MYG\_HUMAN Myoglobin

Score = 51.2 bits (121), Expect = 1e-07,

Identities = 38/146 (26%), Positives = 58/146 (39%), Gaps = 6/146 (4%)

Query 2 LSPADKTNVKAAGKVGAGAHEYGAELERMFLSFPTTKTYFPF-----DLSHGSAQV 55  
      LS + V WGKV A +G E L R+F P T F F D S +

Sbjct 3 LSDGEWQLVLNWGKVEADIPGHGQEVLIRLFKGHPETLEKFDKEKHLKSEDEMKAEDL 62

Query 56 KGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPA 115  
      K HG V AL + + L+ HA K ++ + +S C++ L + P

Sbjct 63 KKHGATVLTALGGILKKKGHHEAEIKPLAQSHATKHKIPVKYLEFISECIIQVLQSKHPG 122

Query 116 EFTPASVHASLDKFLASVSTVLTSKYR 141  
      +F           +++K L           + S Y+

Sbjct 123 DFGADAQGAMNKALELFRKDMASNYK 148

# Not similar sequences

Query: HBA\_HUMAN Hemoglobin alpha subunit

Sbjct: SPAC869.02c [Schizosaccharomyces pombe]

Score = 33.1 bits (74), Expect = 0.24

Identities = 27/95 (28%), Positives = 50/95 (52%), Gaps = 10/95 (10%)

Query 30 ERMFLSFPTTKTYFPHFDSLHGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAAH 89  
++M ++P P+F+ +H + + +A AL N ++DD+ +LSA D

Sbjct 59 QKMLGNYPEV---LPYFNKAHQISL--SQPRILAFALLNYAKNIDDL-TSLSAFMDQIVV 112

Query 90 K---LRVDPVNFKLLSHCLLVTLAAHLPAEF-TPA 120

K L++ ++ ++ HCLL T+ LP++ TPA

Sbjct 113 KHVGLQIKAEHYPIVGHCLLSTMQELLPSDVATPA 147

# Blast Versions

Program	Database	Query
BLASTN	Nucleotide	Nucleotide
BLASTP	Protein	Protein
BLASTX	Protein	Nucleotide translated into protein
TBLASTN	Nucleotide translated into protein	Protein
TBLASTX	Nucleotide translated into protein	Nucleotide translated into protein

# NCBI Blast

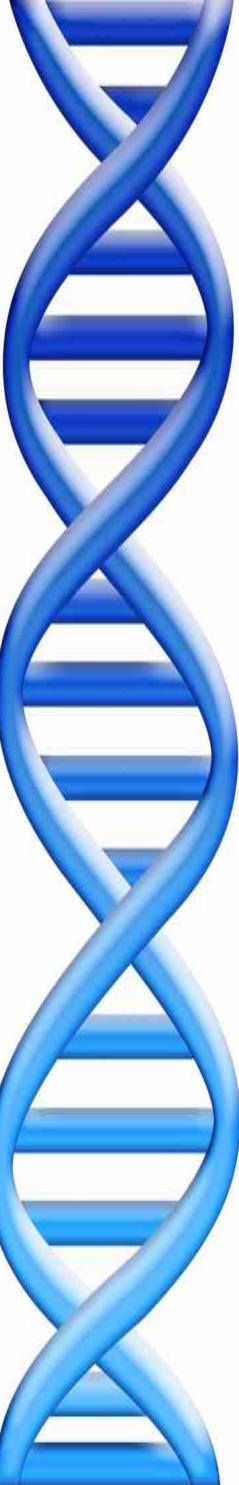
The screenshot shows the NCBI BLAST Basic Local Alignment Search Tool interface. At the top, there's a navigation bar with links for Home, Recent Results, Saved Strategies, and Help. A search bar contains the query "ncbi blast". Below the navigation bar, there's a banner for "Designing or Testing PCR Primers? Try your search in Primer-BLAST." On the left, under "BLAST Assembled Genomes", a list of species includes Human, Mouse, Rat, Arabidopsis thaliana, Oryza sativa, Bos taurus, Danio rerio, Drosophila melanogaster, Gallus gallus, Pan troglodytes, Microbes, and Apis mellifera. Under "Basic BLAST", there are links for nucleotide blast, protein blast, blastx, tblastn, and tblastx, each with a brief description of the search type. On the right, there's a "News" section with a link to "Align two sequences form.", a tip about the standard BLAST submission form, and a "Tip of the Day" section with a link to "How to do Batch BLAST jobs". At the bottom, there's a "Specialized BLAST" section with a list of options like Primer-BLAST, trace archives, conserved domains, and immunoglobulins.

- Nucleotide Databases

- nr:All Genbank
- refseq: Reference organisms
- wgs:All reads

- Protein Databases

- nr:All non-redundant sequences
- Refseq: Reference proteins



# Outline

1. Alignment to other genomes
2. Prediction aka “Gene Finding”
3. Experimental & Functional Assays



# Bacterial Gene Finding and Glimmer

(also Archaeal and viral gene finding)

Arthur L. Delcher and Steven Salzberg  
Center for Bioinformatics and Computational Biology  
Johns Hopkins University

# Genetic Code

		Second letter					
		U	C	A	G		
First letter	U	UUU } Phe UUC } UUA } Leu UUG }	UCU } UCC } Ser UCA } UCG }	UAU } Tyr UAC } <b>UAA Stop</b> <b>UAG Stop</b>	UGU } Cys UGC } <b>UGA Stop</b> UGG Trp	U C A G	Third letter
	C	CUU } CUC } Leu CUA } CUG }	CCU } CCC } Pro CCA } CCG }	CAU } His CAC } CAA } Gln CAG }	CGU } CGC } Arg CGA } CGG }	U C A G	
	A	AUU } AUC } Ile AUA } <b>AUG Met</b>	ACU } ACC } ACA } ACG }	AAU } Asn AAC } AAA } Lys AAG }	AGU } Ser AGC } AGA } Arg AGG }	U C A G	
	G	GUU } GUC } Val GUA } GUG }	GCU } GCC } Ala GCA } GCG }	GAU } Asp GAC } GAA } Glu GAG }	GGU } GGC } Gly GGA } GGG }	U C A G	

- Start:
- AUG
- Stop:
- UAA
  - UAG
  - UGA

# Step One

- Find open reading frames (ORFs).

A diagram illustrating an open reading frame (ORF) in a DNA sequence. The sequence is shown as a horizontal line of colored boxes representing nucleotides: red for Adenine (A), yellow for Thymine (T), and black for Guanine (G). A green rectangular box highlights a segment of the sequence: ...TAGATGAATGGCTCTTTAGATAAAATTTCATGAAAAAATTGA.... An arrow labeled "Start codon" points to the first three nucleotides, TAG. Another arrow labeled "Stop codon" points to the last three nucleotides, ATT, which is a TAA stop codon. The sequence continues beyond the highlighted region.

Start codon

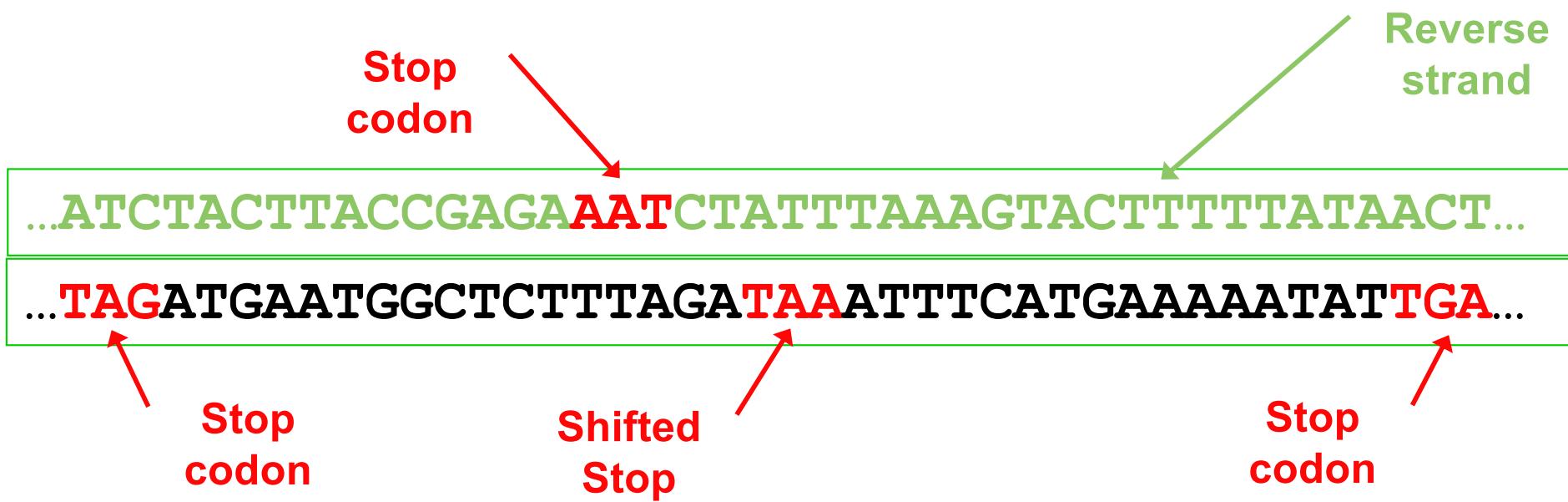
Stop codon

Stop codon

...TAGATGAATGGCTCTTTAGATAAAATTTCATGAAAAAATTGA...

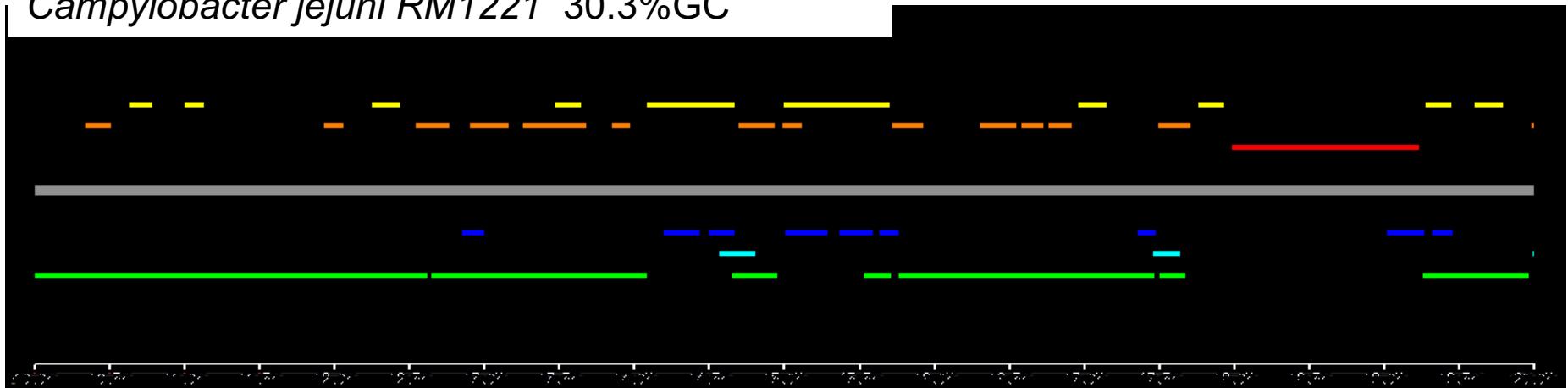
# Step One

- Find open reading frames (ORFs).



- But ORFs generally overlap ...

*Campylobacter jejuni RM1221* 30.3%GC

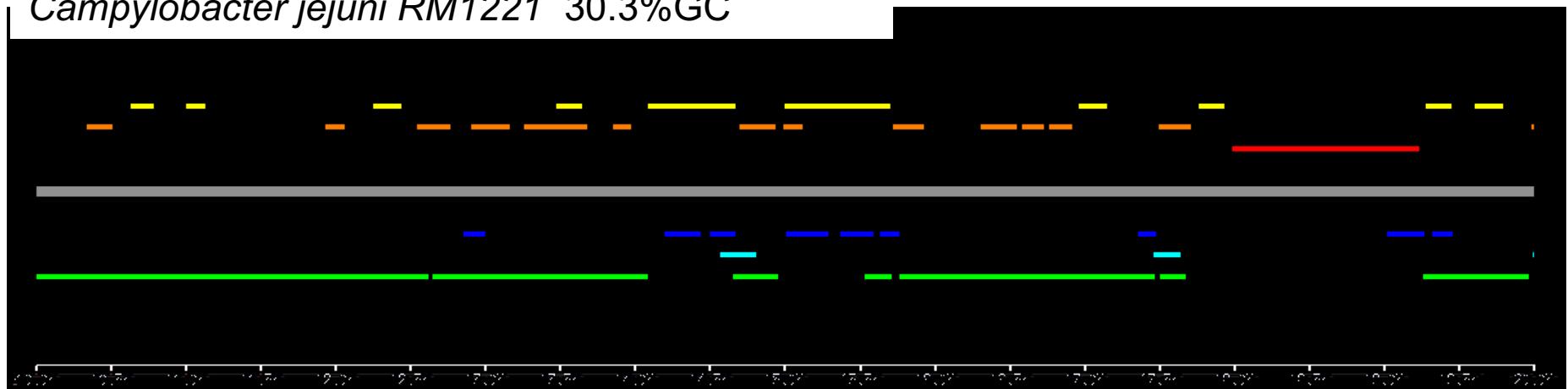


All ORFs longer than 100bp on both strands shown  
- color indicates reading frame  
Longest ORFs likely to be protein-coding genes

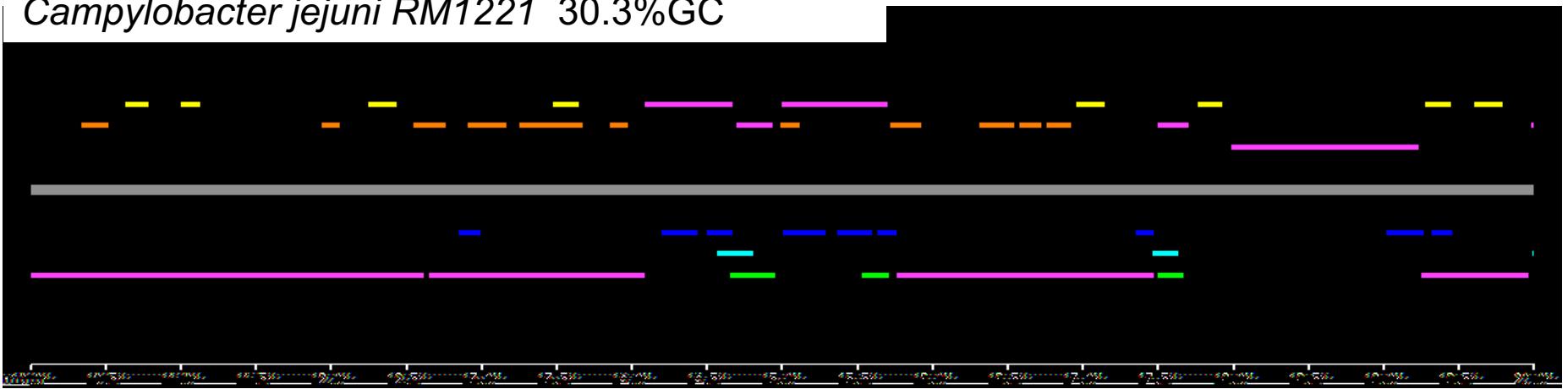
Note the low GC content

All genes are ORFs but not all ORFs are genes

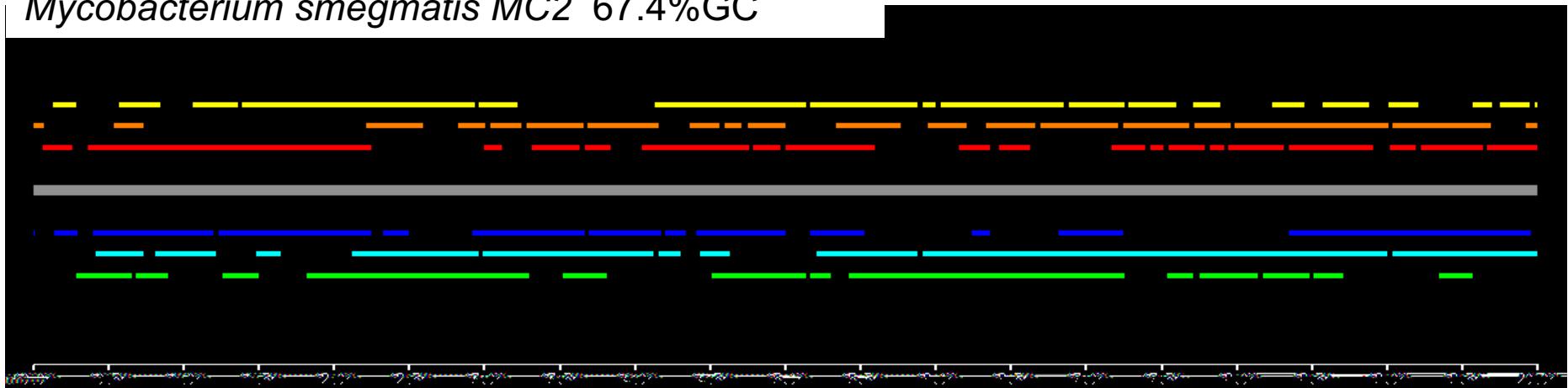
*Campylobacter jejuni* RM1221 30.3%GC



*Campylobacter jejuni* RM1221 30.3%GC

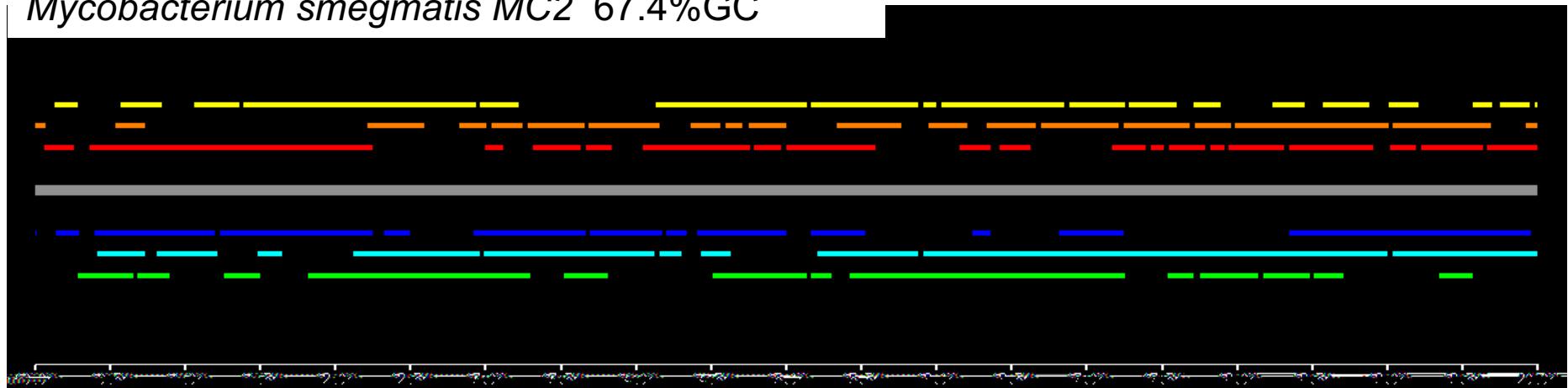


*Mycobacterium smegmatis MC2* 67.4%GC

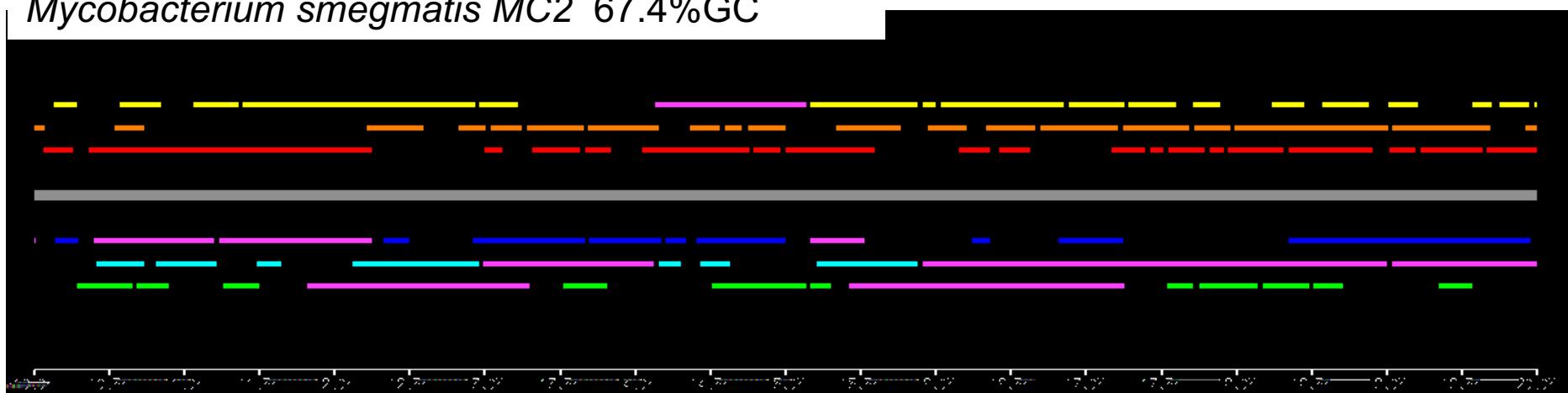


Note what happens in a high-GC genome

*Mycobacterium smegmatis MC2* 67.4%GC



*Mycobacterium smegmatis MC2* 67.4%GC

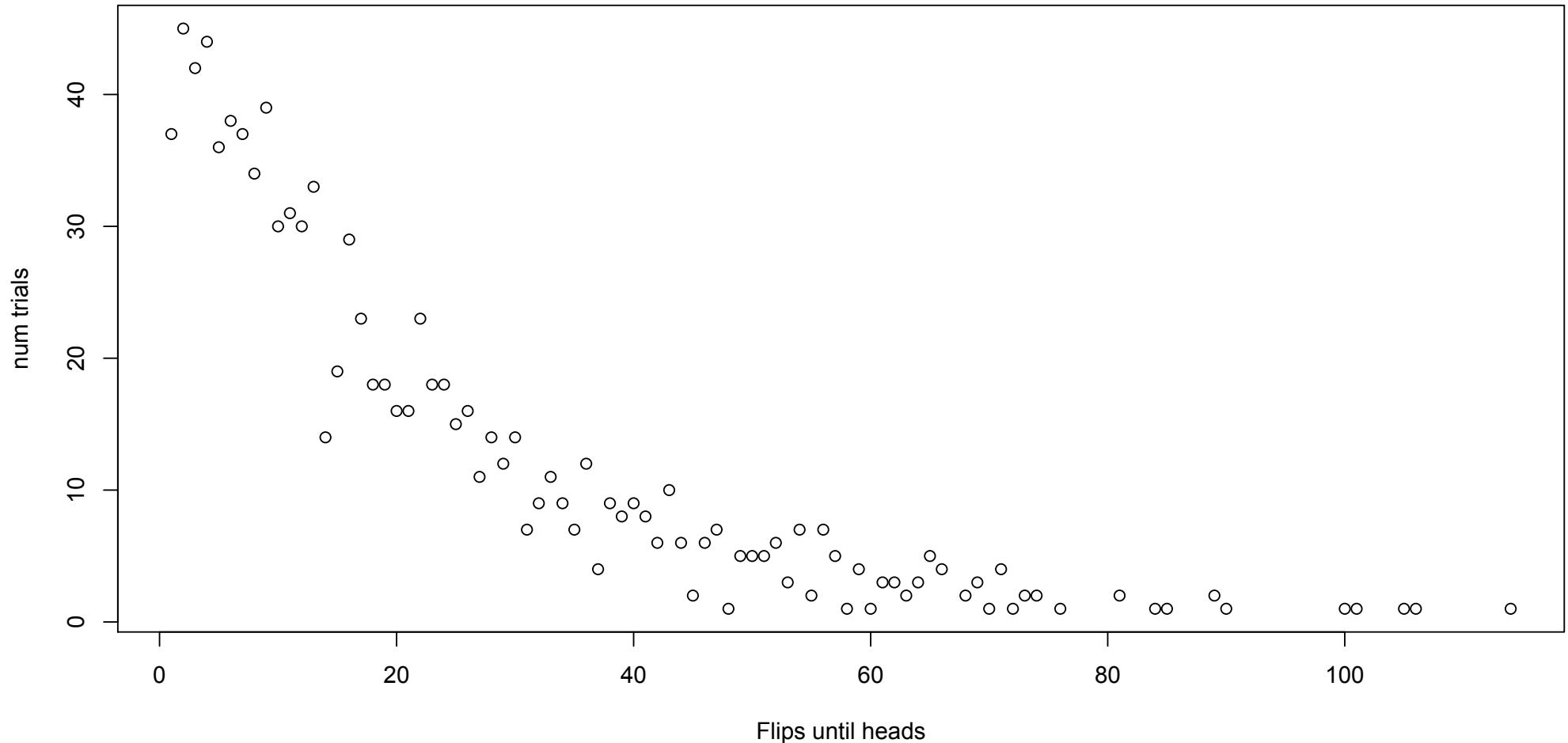




# Flipping a Biased Coin

$$P(\text{heads}) = 61/64 (95.4\%) \quad P(\text{tails}) = 3/64 (4.6\%)$$

How many flips until my first tail?

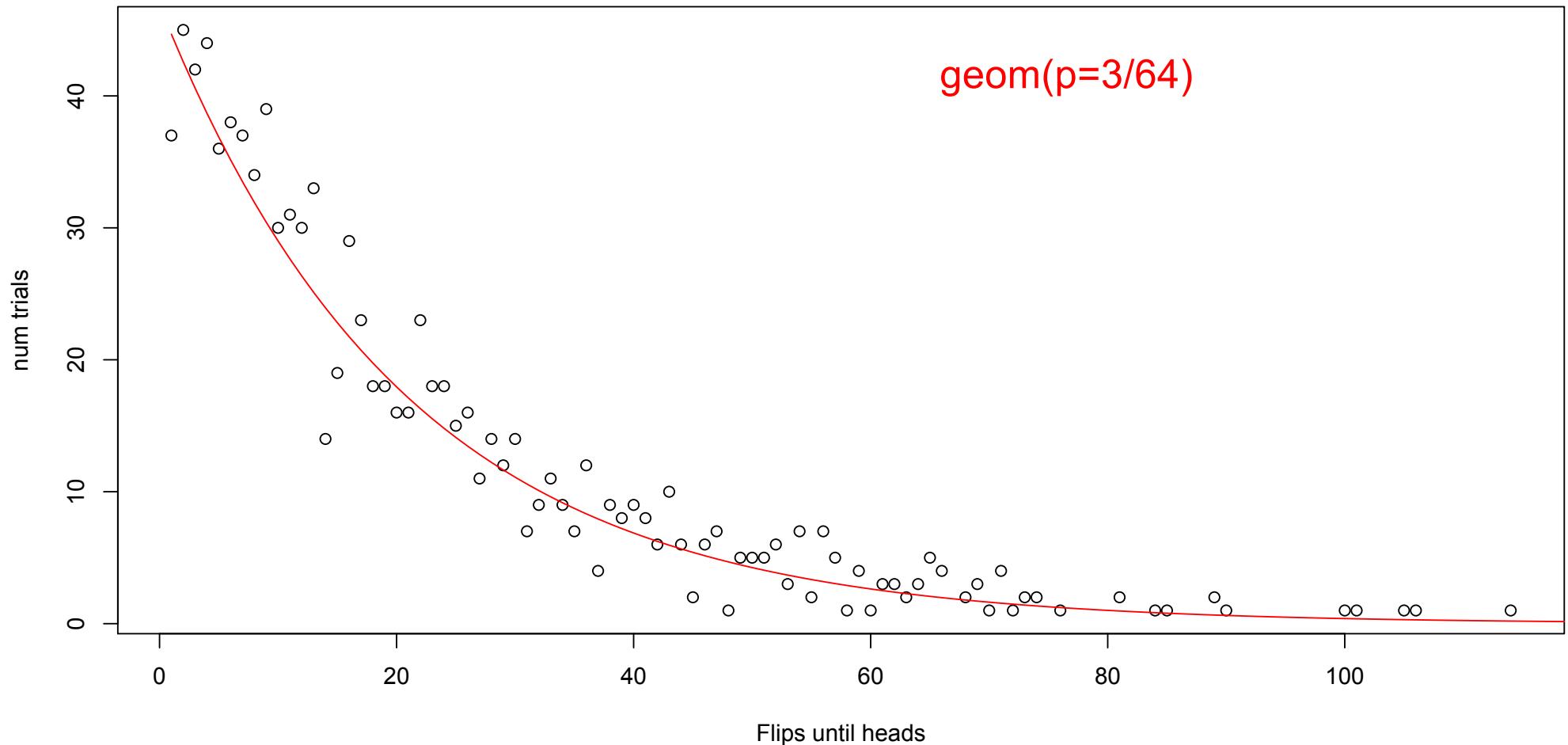


# Flipping a Biased Coin

$$P(\text{heads}) = 61/64 (95.4\%) \quad P(\text{tails}) = 3/64 (4.6\%)$$

How many flips until my first tail?

Geometric Distribution:  $P(X=x) = p_{\text{heads}}^{x-1} p_{\text{tails}}$

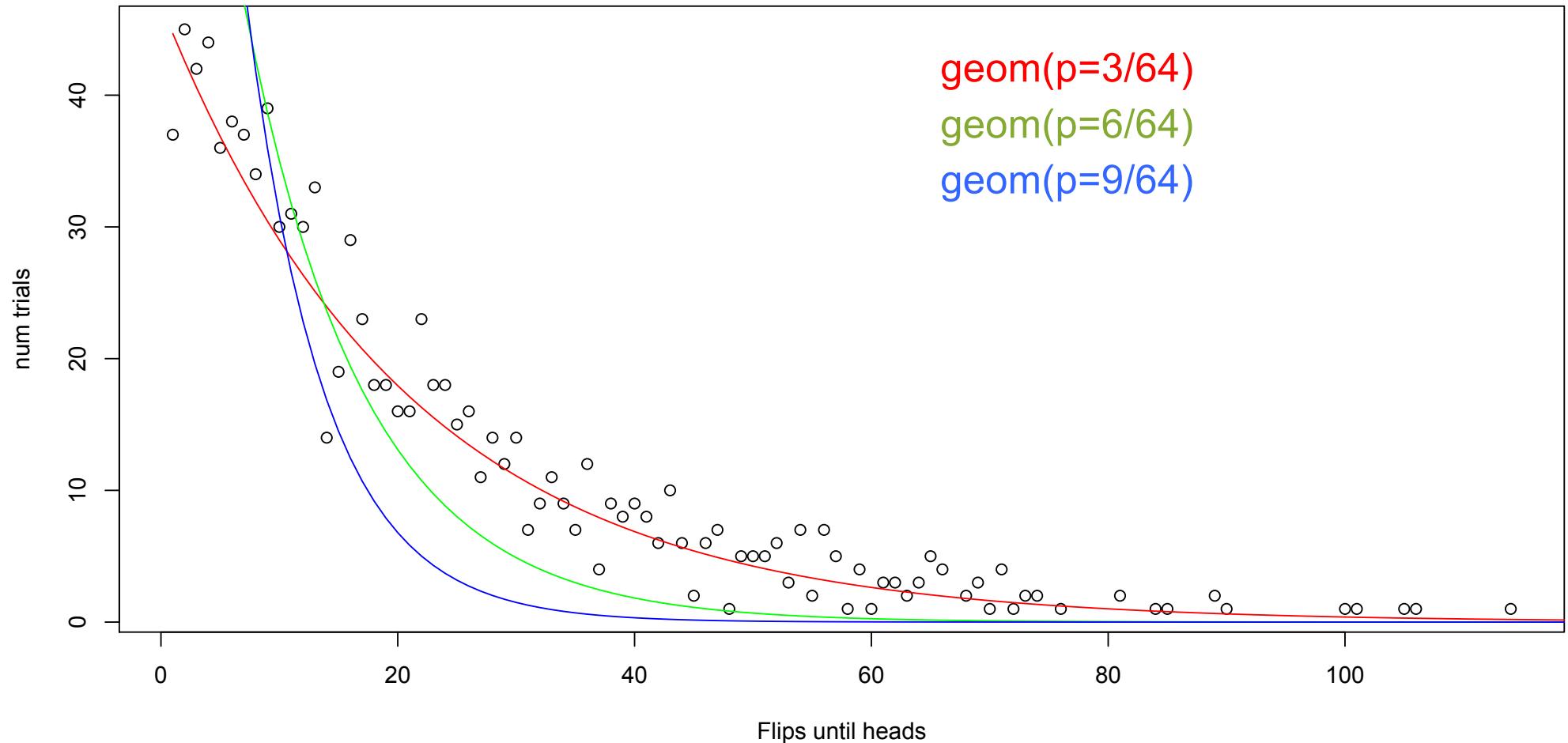


# Flipping a Biased Coin

$$P(\text{heads}) = 61/64 (95.4\%) \quad P(\text{tails}) = 3/64 (4.6\%)$$

How many flips until my first tail?

Geometric Distribution:  $P(X=x) = p_{\text{heads}}^{x-1} p_{\text{tails}}$

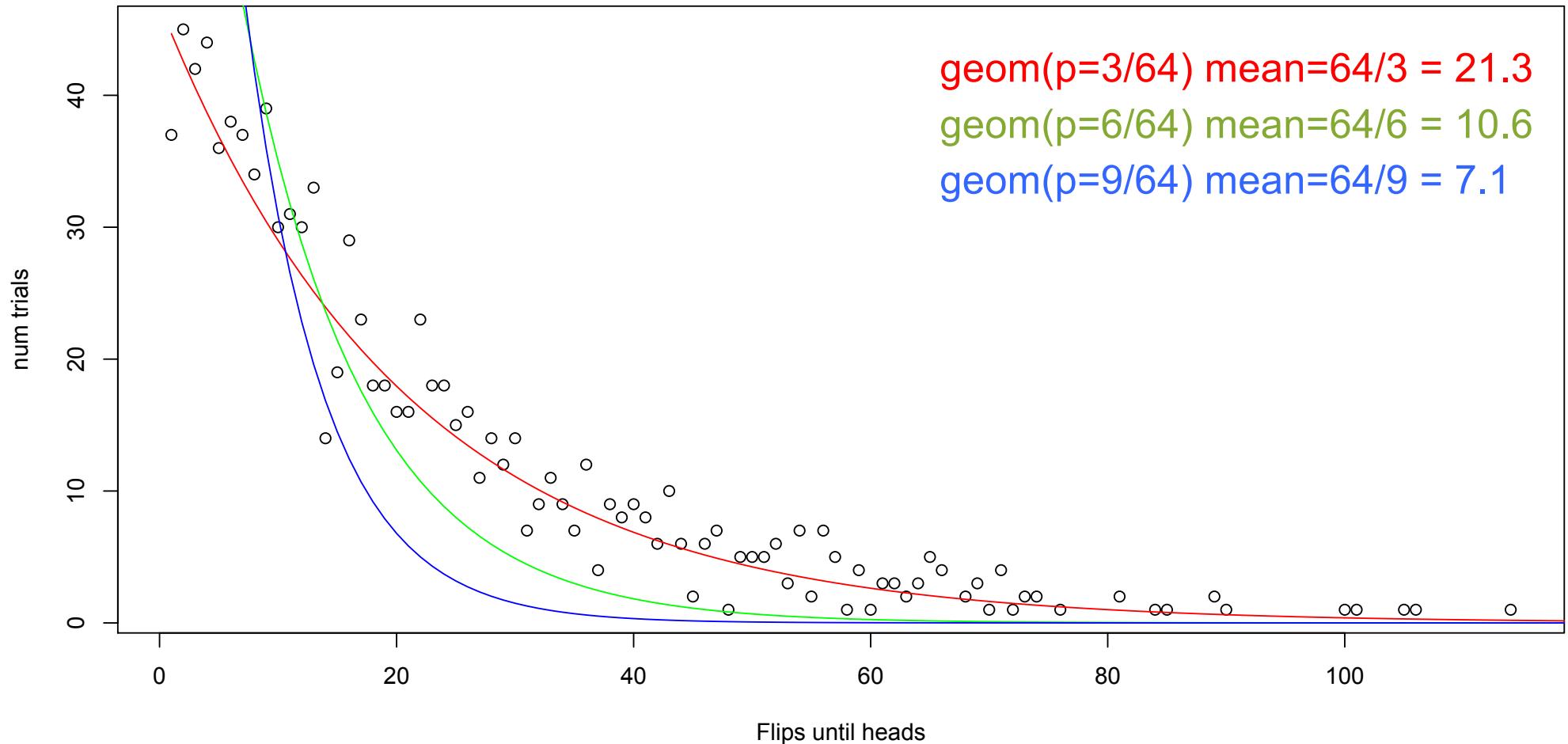


# Flipping a Biased Coin

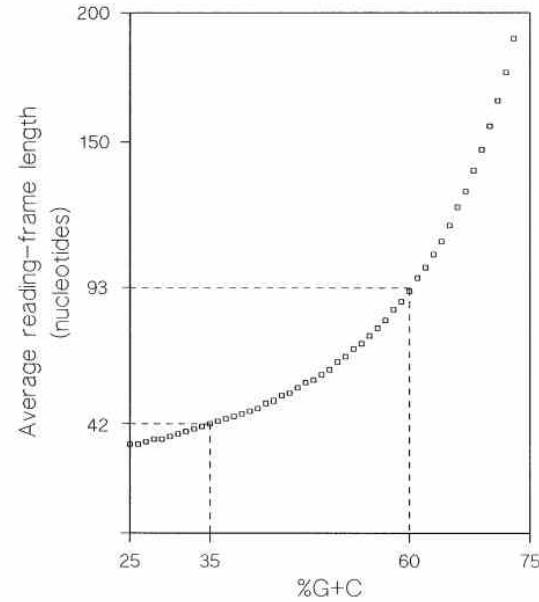
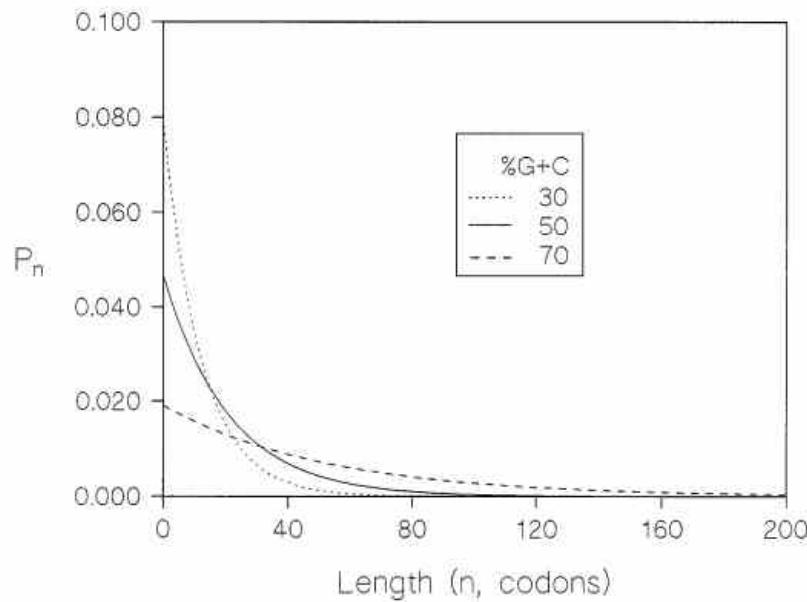
$$P(\text{heads}) = 61/64 (95.4\%) \quad P(\text{tails}) = 3/64 (4.6\%)$$

How many flips until my first tail?

Geometric Distribution:  $P(X=x) = p_{\text{heads}}^{x-1} p_{\text{tails}}$



# Stop Codon Frequencies



**If the sequence is mostly A+T, then likely to form stop codons by chance!**

**In High A+T (Low G+C):**

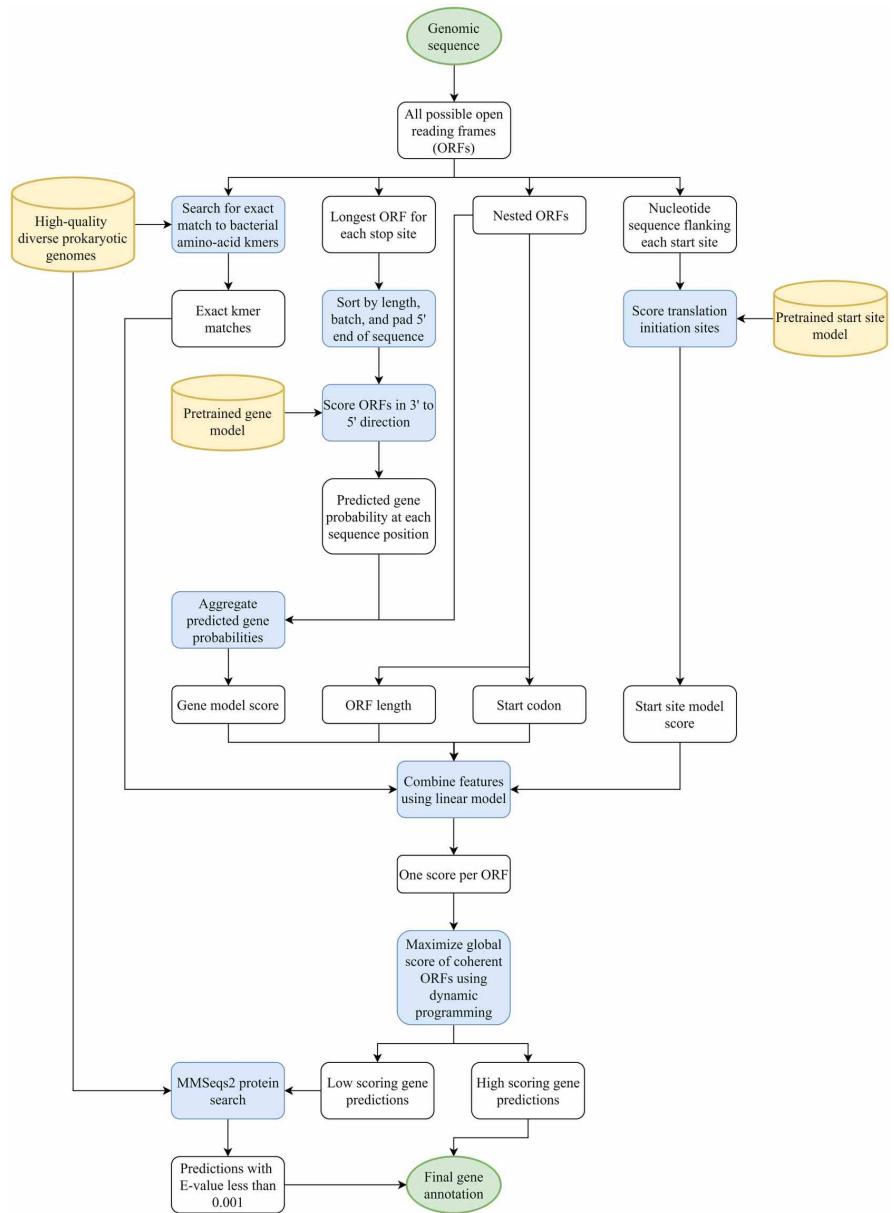
Frequent stop codons; Short Random ORFs; long ORFs likely to be true genes

**In High G+C (Low A+T):**

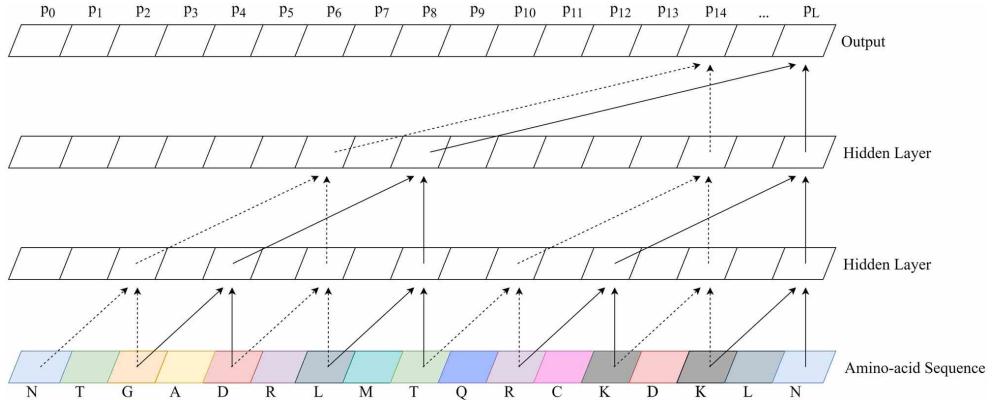
Rare stop codons; Long Random ORFs; harder to identify true genes

**A relationship between GC content and coding-sequence length.**

Oliver & Marín (1996) J Mol Evol. 43(3):216-23.



## Temporal Convolutional Network



**Balrog: A universal protein model for prokaryotic gene prediction**

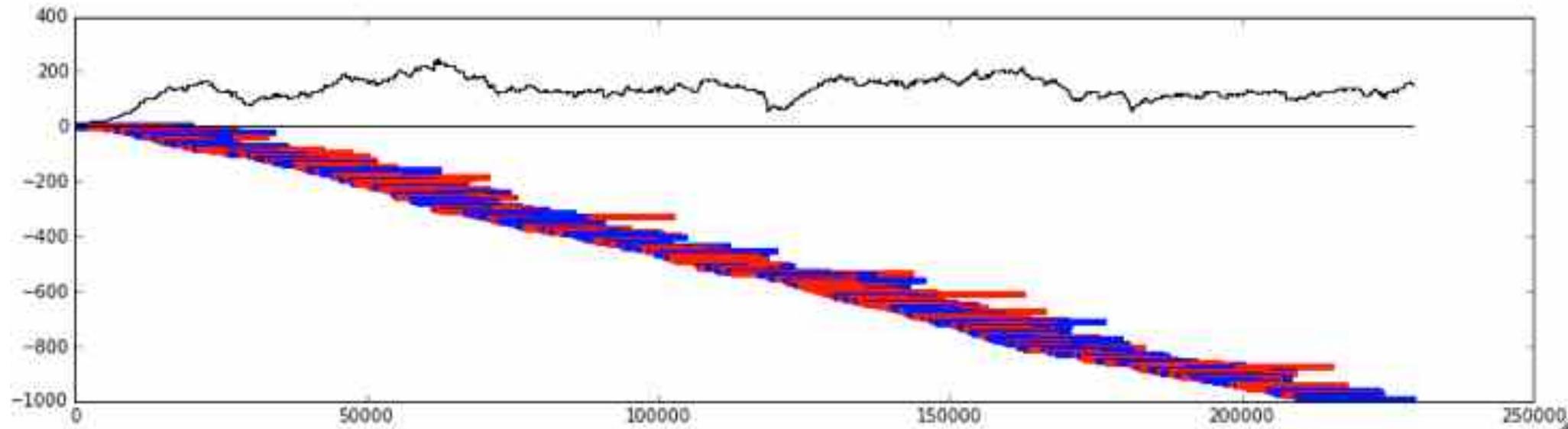
Sommer, MJ, Salzberg, SL (2021) PLOS Comp. Bio. doi: 10.1371/journal.pcbi.1008727

# Probabilistic Methods

- Create models that have a probability of generating any given sequence.
  - Evaluate gene/non-genome models against a sequence
- Train the models using examples of the types of sequences to generate.
  - Use RNA sequencing, homology, or “obvious” genes
- The “score” of an orf is the probability of the model generating it.
  - Most basic technique is to count how kmers occur in known genes versus intergenic sequences
  - More sophisticated methods consider variable length contexts, “wobble” bases, other statistical clues

# Plane Sweep Algorithm

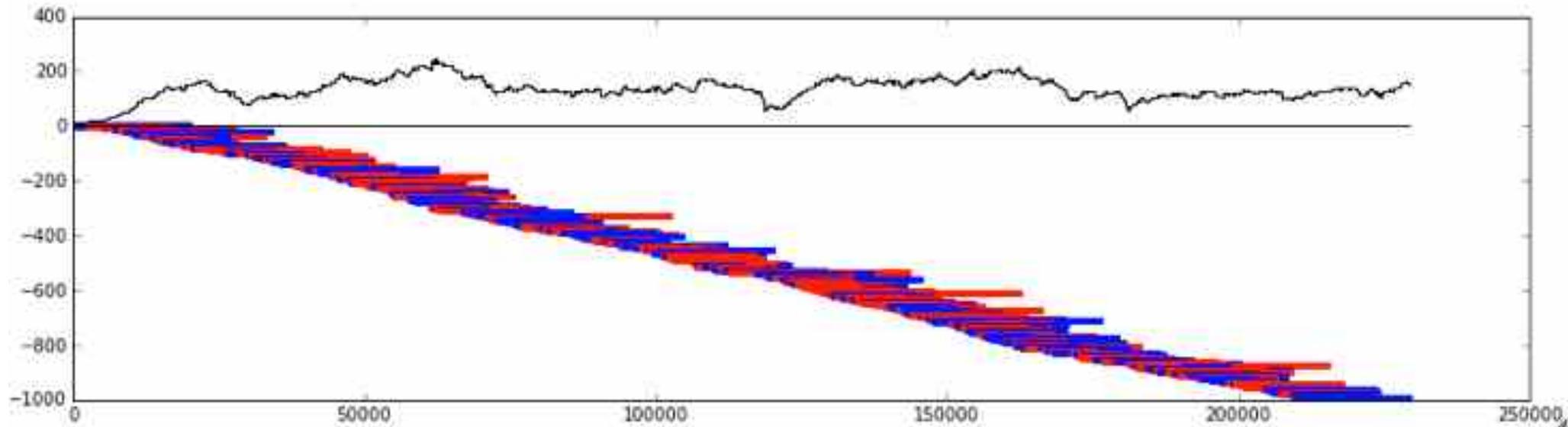
# Coverage across the genome



```
$ head -3 ~/readid.start.stop.txt  
1 0 19814  
2 799 19947  
3 1844 13454
```

```
$ tail -3 ~/readid.start.stop.txt  
1871 973590 965902  
1872 966703 973521  
1873 973632 966946
```

# Coverage across the genome



```
print "Plotting layout"

## draw the layout of reads
for i in xrange(min(MAX_READS_LAYOUT, len(reads))):
    r = reads[i]
    readid = r[0]
    start  = r[1]
    end    = r[2]
    rc     = r[3]
    color  = "blue"

    if (rc == 1):
        color = "red"

    plt.plot ([start,end], [-2*i, -2*i], lw=4, color=color)
```



r[1] is start pos  
r[2] is end pos

# Brute Force Coverage Profile

```
print "Brute force computing coverage over %d bp" % (totallen)

starttime = time.time()
brutecov = [0] * totallen

for r in reads:
    # print " -- [%d, %d]" % (r[1], r[2])

    for i in xrange(r[1], r[2]):
        brutecov[i] += 1

brutetime = (time.time() - starttime) * 1000.0

print " Brute force complete in %.02f ms" % (brutetime)
print brutecov[0:10]
```

```
Brute force computing coverage over 973898 bp
Brute force complete in 4435.00 ms
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Notice that it took 4435 ms for this to complete

Add 1 to coverage vector at every position the read covers

# Delta Encoding

## aka run length encoding

```
deltacov = []
curcov = -1
for i in xrange(0, len(brutecov)):
    if brutecov[i] != curcov:
        curcov = brutecov[i]
        delta = (i, curcov)
        deltacov.append(delta)

## Finish up with the last position
deltacov.append((totallen, 0))
```

Only record those positions when the coverage changes

Delta encoding coverage plot  
Delta encoding required only 3697 steps, saving 99.62% of the space in 151.32 ms

```
0: [0,1]
1: [799,2]
2: [1844,3]
...
3694: [973770,2]
3695: [973779,1]
3696: [973898,0]
```

# Plot Coverage and Read Positions

```
## expand the coverage profile by this amount so that it is easier to see
YSCALE = 5

## draw the layout of reads
for i in xrange(min(MAX_READS_LAYOUT, len(reads))):
    r = reads[i]
    readid = r[0]
    start = r[1]
    end = r[2]
    rc = r[3]
    color = "blue"

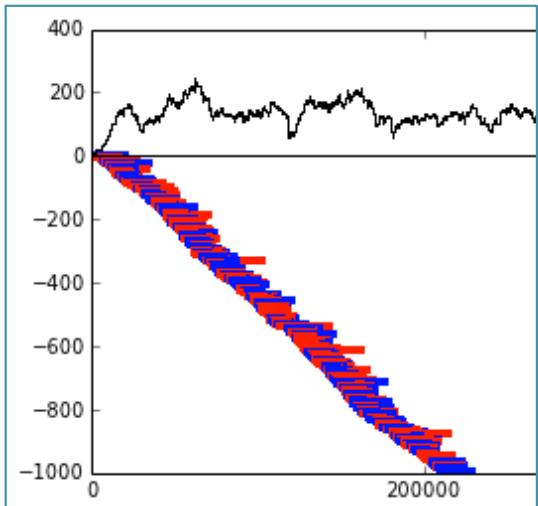
    if (rc == 1):
        color = "red"

    plt.plot ([start,end], [-2*i, -2*i], lw=4, color=color)

## draw the base of the coverage plot
plt.plot([0, totallen], [0,0], color="black")

## draw the coverage plot
for i in xrange(len(deltacov)-1):
    x1 = deltaxcov[i][0]
    x2 = deltaxcov[i+1][0]
    y1 = YSCALE*deltacov[i][1]
    y2 = YSCALE*deltacov[i+1][1]

    ## draw the horizontal line
    plt.plot([x1, x2], [y1, y1], color="black")
    ## and now the right vertical to the new coverage level
    plt.plot([x2, x2], [y1, y2], color="black")
```



Plot Each Read



Plot Each  
Coverage Step



# Plot Coverage and Read Positions

```
## expand the coverage profile by this amount so that it is easier to see  
YSCALE = 5  
  
## draw the layout of reads  
for i in xrange(min(MAX_READS_LAYOUT, len(reads))):  
    r = reads[i]  
    readid = r[0]  
    start = r[1]  
    end = r[2]  
    rc = r[3]  
    color = "blue"  
  
    if (rc == 1):  
        color = "red"  
  
    plt.plot ([start,end], [-2*i, -2*i], lw=4, color=color)
```

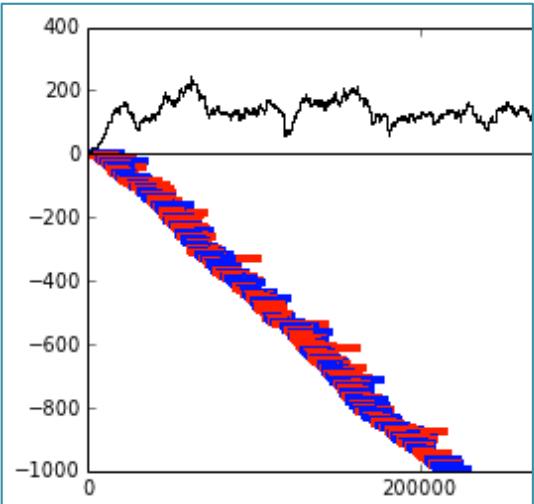
```
## draw the base of the coverage plot  
plt.plot([0, totallen], [0,0], color="black")
```

```
## draw the coverage plot  
for i in xrange(len(deltacov)):  
    x1 = deltaxcov[i][0]  
    x2 = deltaxcov[i+1][0]  
    y1 = YSCALE*deltacov[i][1]  
    y2 = YSCALE*deltacov[i+1][1]
```

Brute Force works, but is pretty slow.

How can we make it go faster?

```
## draw the horizontal line  
plt.plot([x1, x2], [y1, y1], color="black")  
  
## and now the right vertical to the new coverage level  
plt.plot([x2, x2], [y1, y2], color="black")
```



Plot Each Read



Plot Each Coverage Step



# Plane Sweep

pos:	1	5	10	15	20	25	30	35	40	45	50	55
pos:												
r1:	[=====]											
r2:	[=====]											
r3:	[=====]											
r4:	[=====]											
r5:	[=====]											

## ***The basic algorithm works like this:***

- Assume layout is in sorted order by start position (or explicitly sort by start position)
- use a “list” to track how many reads currently intersect the plane keyed by end coord
  - the number of elements in the list corresponds to the current depth
- walking from start position to start position
  - check to see if we past any read ends
  - coverage goes down by one when a read ends
  - coverage goes up by one when new read is encountered

# Plane Sweep

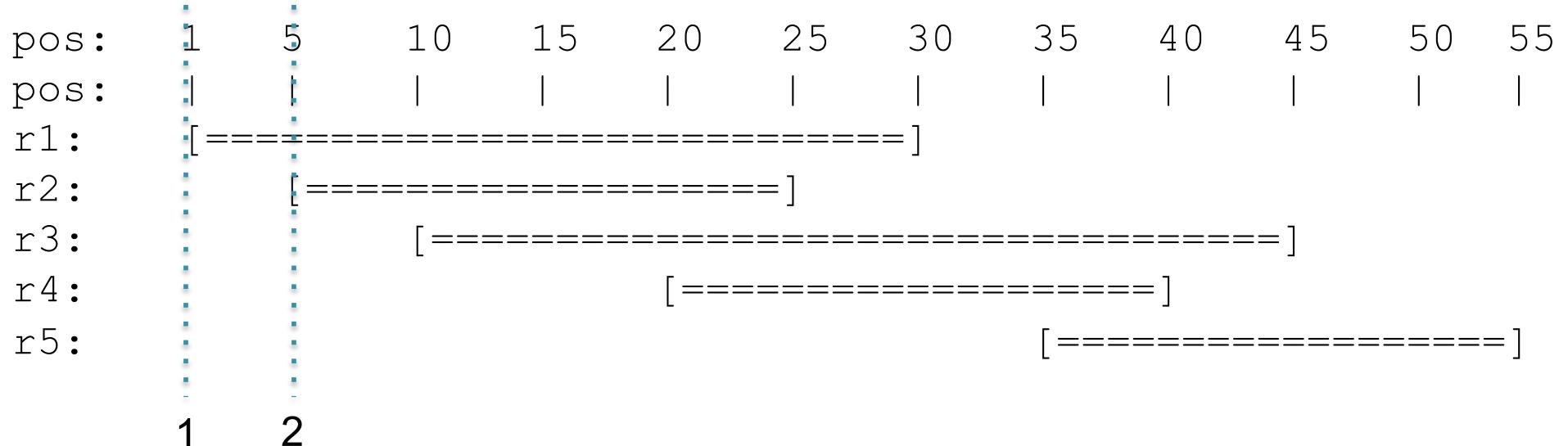
pos:	1	5	10	15	20	25	30	35	40	45	50	55
pos:												
r1:	[=====]											
r2:	[=====]											
r3:	[=====]											
r4:	[=====]											
r5:	[=====]											
	1											

*arrive at r1 [1,30]:*

**active set is empty; add to active set: 30**

**output (1,1)**

# Plane Sweep



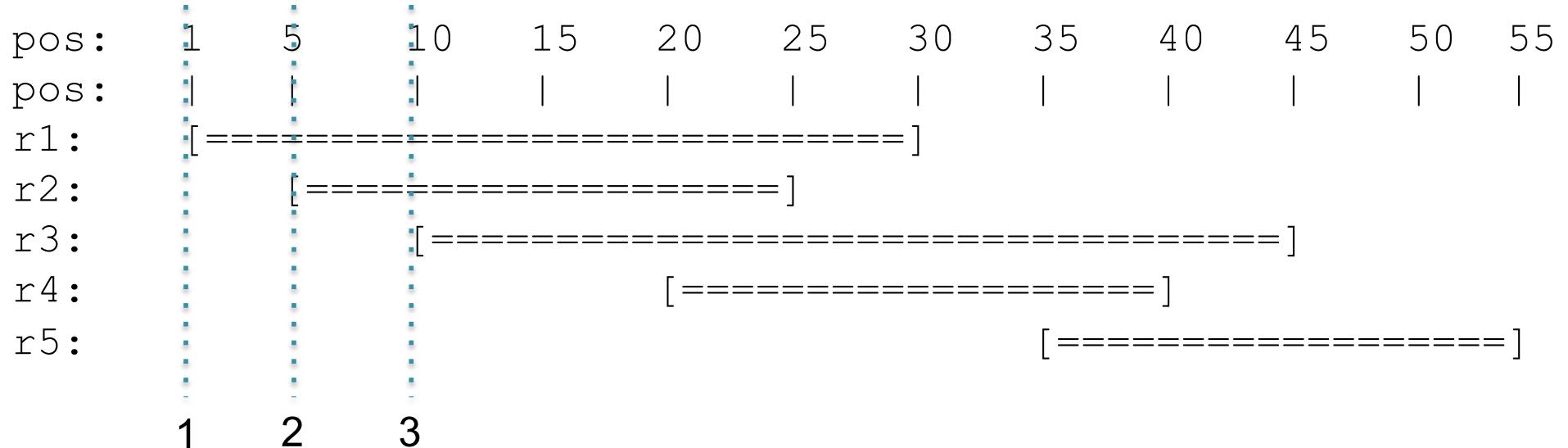
***arrive at r1 [1,30]:***

***active set is empty; add to active set: 30  
output (1,1)***

***arrive at r2 [5,25]:***

***5 < 30: add to active set: 25, 30 <- notice insert out of order  
output (5, 2)***

# Plane Sweep



**arrive at r1 [1,30]:**

**active set is empty; add to active set: 30  
output (1,1)**

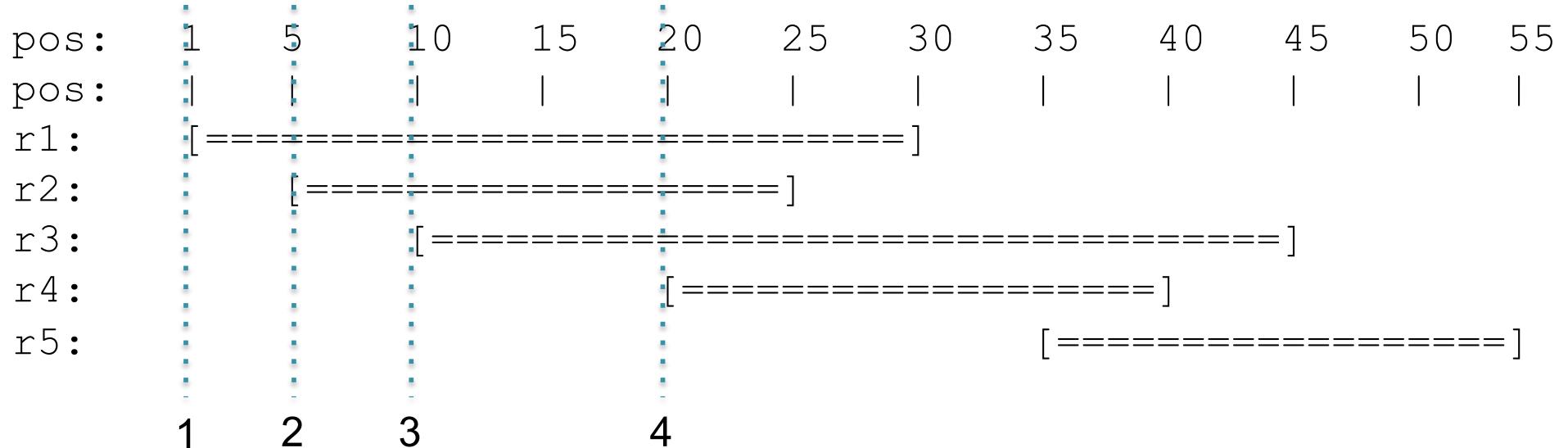
**arrive at r2 [5,25]:**

**5 < 30: add to active set: 25, 30 <- notice insert out of order  
output (5, 2)**

**arrive at r3 [10,45]:**

**10 < 25; add to active set: 25, 30, 45  
output (10, 3)**

# Plane Sweep



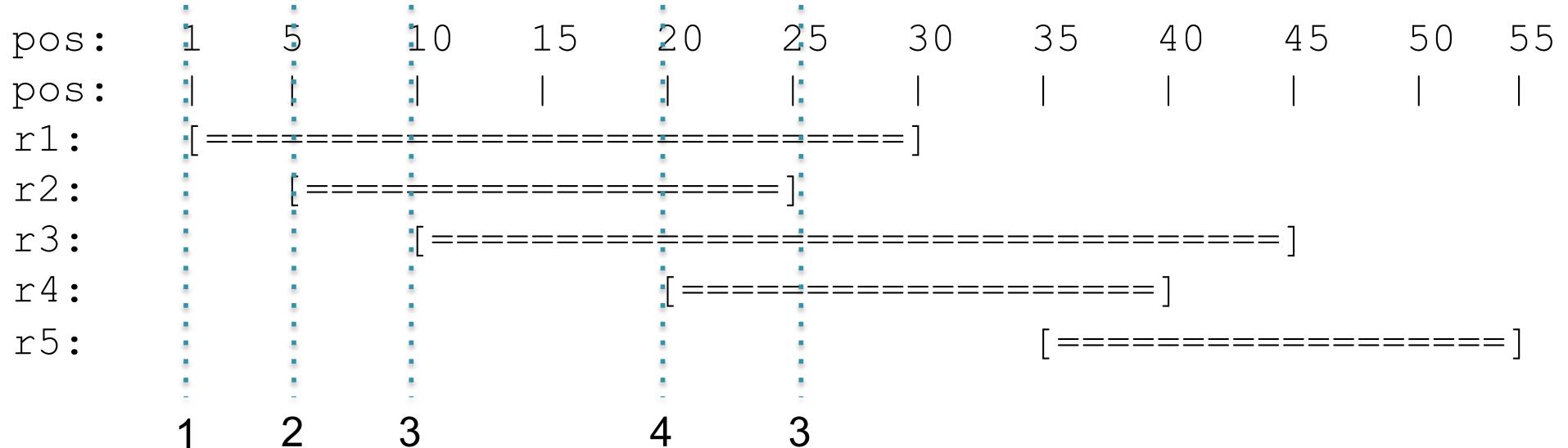
*arrive at r3 [10,45]:*

**$10 < 25$ ; add to active set: 25, 30, 45**  
**output (10, 3)**

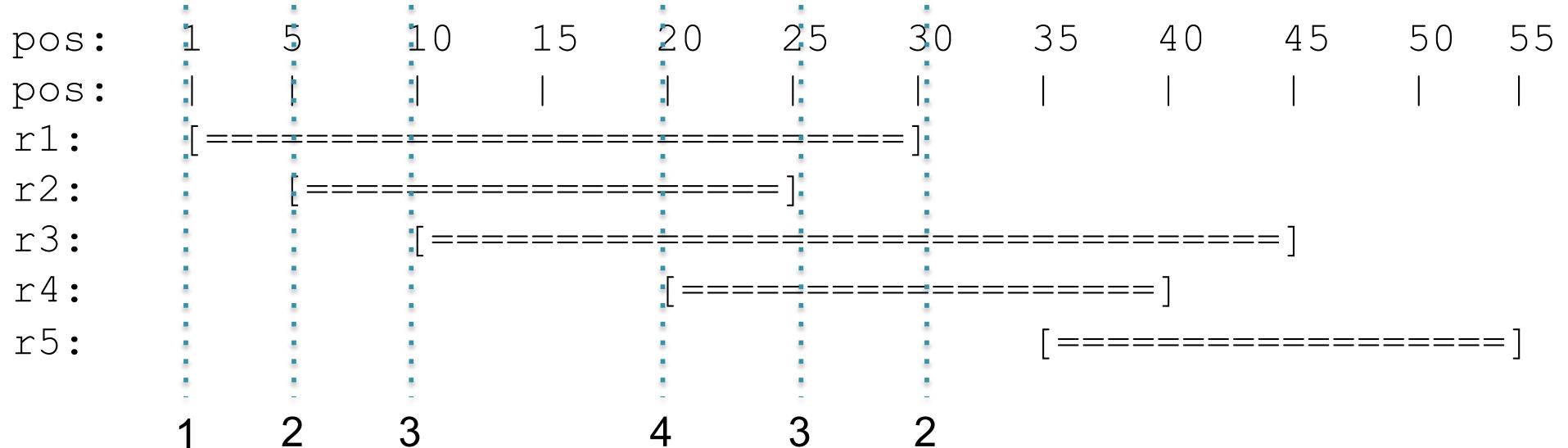
*arrive at r4 [20,40]:*

**$20 < 25$ ; add to active set: 25, 30, 40, 45 <- out of order again**  
**output (20, 4)**

# Plane Sweep



# Plane Sweep



*arrive at r5[35,55]:*

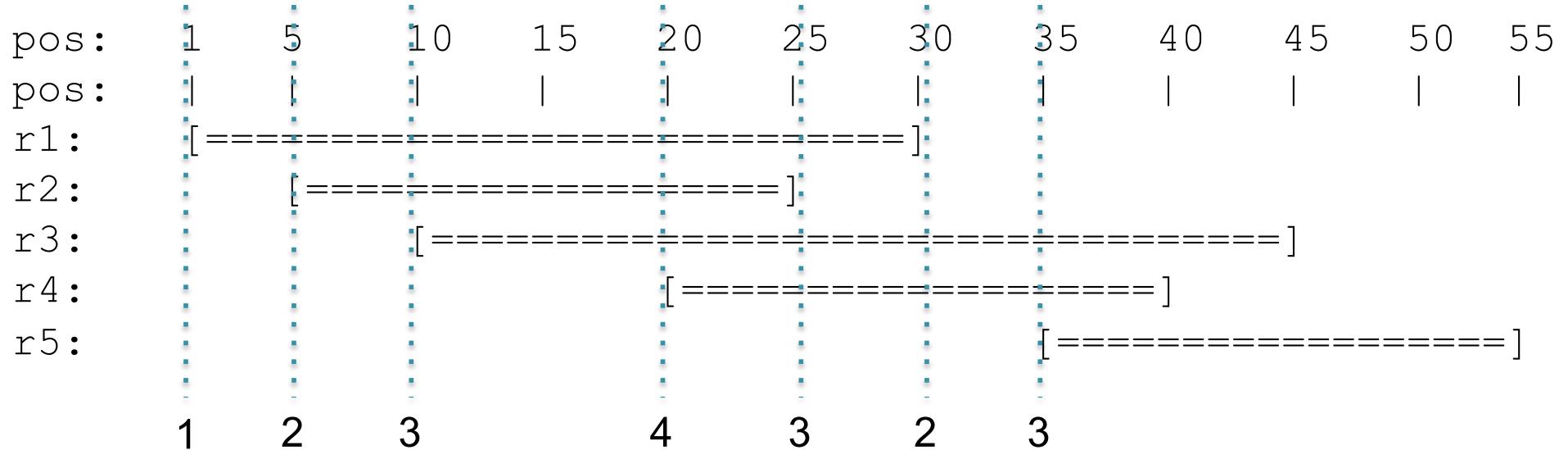
*35 > 25: step down at 25; active set: 30, 40, 45*

*output (25, 3)*

*35 > 30: step down at 30; active set: 40, 45*

*output (30, 2)*

# Plane Sweep



*arrive at r5[35,55]:*

*35 > 25: step down at 25; active set: 30, 40, 45*

*output (25, 3)*

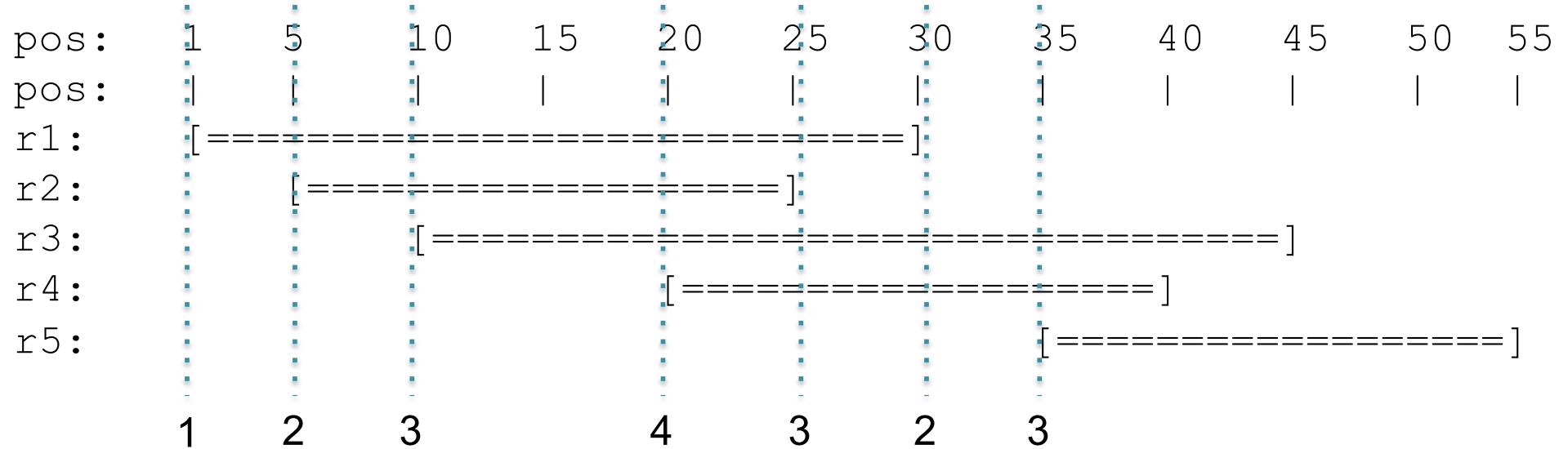
*35 > 30: step down at 30; active set: 40, 45*

*output (30, 2)*

*35 < 40: add to active set: 40, 45, 55*

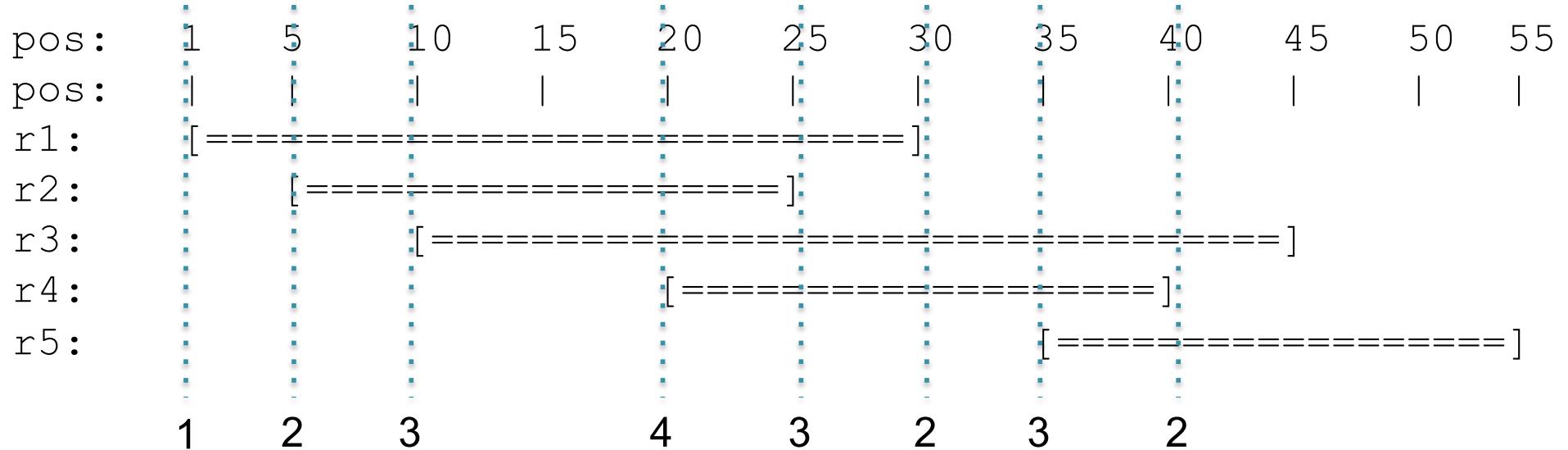
*output (35, 3)*

# Plane Sweep



*Flush:*

# Plane Sweep

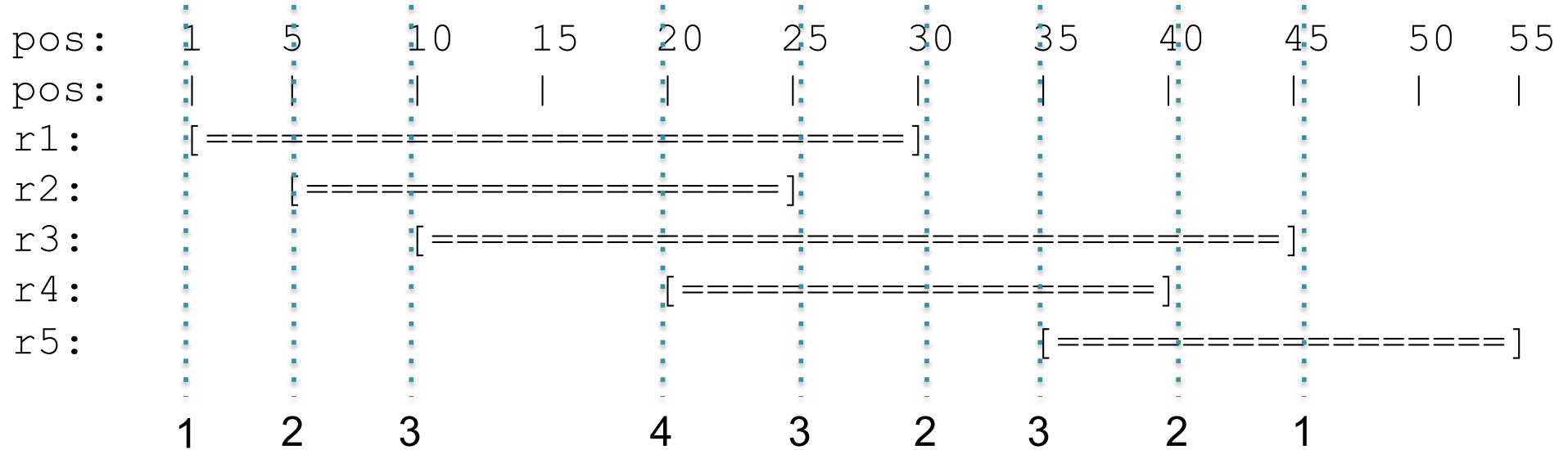


**Flush:**

**step down at 40; active set: 45, 55**

**output (40, 2)**

# Plane Sweep



**Flush:**

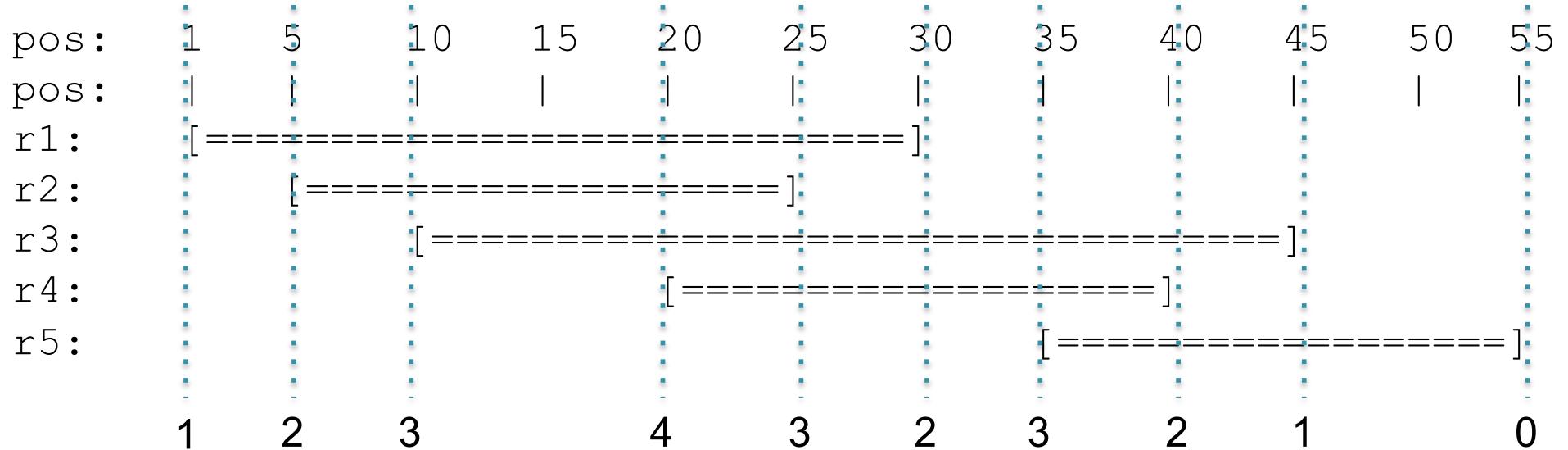
***step down at 40; active set: 45, 55***

***output (40, 2)***

***step down at 45: active set: 55***

***output (45, 1)***

# Plane Sweep



**Flush:**

**step down at 40; active set: 45, 55**

**output (40, 2)**

**step down at 45: active set: 55**

**output (45, 1)**

**step down at 55: active set: {}**

**output (55, 0)**

# Plane Sweep

```
## record the delta encoded depth using a plane sweep
deltacovplane = []

## use a list to record the end positions of the elements currently in plane
planelist = []

## BEGIN SWEEP
for r in reads:
    startpos = r[1]
    endpos   = r[2]

    ## clear out any positions from the plane that we have already moved past
    while (len(planelist) > 0):

        if (planelist[0] <= startpos):
            ## the coverage steps down, extract it from the front of the list
            oldend = planelist.pop(0)
            deltacovplane.append((oldend, len(planelist)))
        else:
            break

    ## Now insert the current endpos into the correct position into the list
    insertpos = -1
    for i in xrange(len(planelist)):
        if (endpos < planelist[i]):
            insertpos = i
            break

    if (insertpos > 0):
        planelist.insert(insertpos, endpos)
    else:
        planelist.append(endpos)

    ## Finally record that the coverage has increased
    deltacovplane.append((startpos, len(planelist)))

## Flush any remaining end positions
while (len(planelist) > 0):
    oldend = planelist.pop(0)
    deltacovplane.append((oldend, len(planelist)))
```

Keep track of end positions of reads that have been seen so far

Check to see if any reads have ended before the start of this one

Add the end of the current read to the sweep plane in sorted order

Why sorted?

Beginning list-based plane sweep over 1873 reads

Plane sweep found 3746 steps, saving 99.62% of the space in 48.90 ms (90.69 speedup)!

# Plane Sweep

```
## record the delta encoded depth using a plane sweep
deltacovplane = []

## use a list to record the end positions of the elements currently in plane
planelist = []

## BEGIN SWEEP
for r in reads:
    startpos = r[1]
    endpos   = r[2]

    ## clear out any positions from the plane that we have already moved past
    while (len(planelist) > 0):

        if (planelist[0] <= startpos):
            ## the coverage steps down, extract it from the front of the list
            oldend = planelist.pop(0)
            delta
        else:
            break

        ## Now in
        insertpos
        for i in
            if (end
                ins
                bre
            if (inser
                planelist.insert(insertpos, endpos)
            else:
                planelist.append(endpos)

## Finally record that the coverage has increased
deltacovplane.append((startpos, len(planelist)))

## Flush any remaining end positions
while (len(planelist) > 0):
    oldend = planelist.pop(0)
    deltacovplane.append((oldend, len(planelist)))
```

See notes on how to handle  
reads that have same  
coordinates

(Annoying bookkeeping :-/ )

Keep track of end  
positions of reads  
that have been  
seen so far

Check to see if  
any reads have  
ended before the  
start of this one

Add the end of the  
current read to the  
sweep plane in  
sorted order

Beginning list-based plane sweep over 1873 reads

Plane sweep found 3746 steps, saving 99.62% of the space in 48.90 ms (90.69 speedup)!

# Plane Sweep

```
## record the delta encoded depth using a plane sweep
deltacovplane = []

## use a list to record the end positions of the elements currently in plane
planelist = []

## BEGIN SWEEP
for r in reads:
    startpos = r[1]
    endpos   = r[2]

    ## clear out any positions from the plane that we have already moved past
    while (len(planelist) > 0):

        if (planelist[0] <= startpos):
            ## the coverage steps down, extract it from the front of the list
            oldend = planelist.pop(0)
            deltacovplane.append((oldend, len(planelist)))
        else:
            break

    ## Now insert the current endpos into the correct position into the list
    insertpos = -1
    for i in xrange(len(planelist)):
        if (endpos < planelist[i]):
            insertpos = i
            break

    if (insertpos > 0):
        planelist.insert(insertpos, endpos)
    else:
        planelist.append(endpos)

    ## Finally record that the coverage has increased
    deltacovplane.append((startpos, len(planelist)))

## Flush any remaining end positions
while (len(planelist) > 0):
    oldend = planelist.pop(0)
    deltacovplane.append((oldend, len(planelist)))
```

Keep track of end positions of reads that have been seen so far

Check to see if any reads have ended before the start of this one

Add the end of the current read to the sweep plane in sorted order

Do we really need the whole list to be sorted?

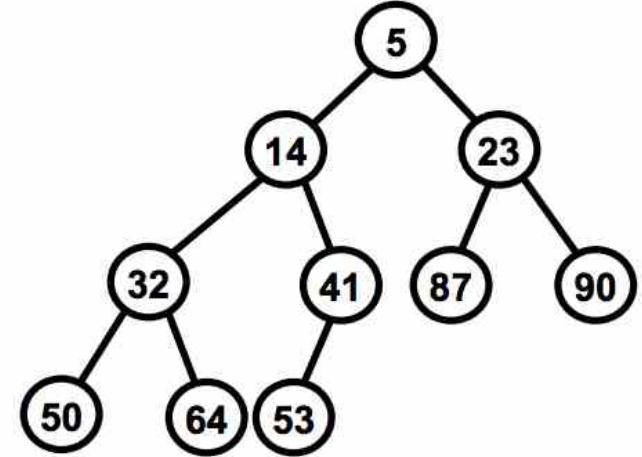
Beginning list-based plane sweep over 1873 reads

Plane sweep found 3746 steps, saving 99.62% of the space in 48.90 ms (90.69 speedup)!

# Heaps & Priority Queues

**Binary Min Heap:** Binary tree such that the value of a node is less than or equal to the value of its 2 children

Similar to a binary search tree, although there are no guarantees about the relationships of the left and right children



Very efficient data structure for dynamically maintaining a set of element while allowing you to find the minimum (or maximum) very fast:

Insert:  $O(\lg(n))$       <- super fast

Remove:  $O(\lg(n))$       <- super fast

Find-min:  $O(1)$       <- instantaneous

Key to fast performance derives from ***heap shape property***: the tree is guaranteed to be a complete binary tree, meaning it will remain balanced and the height will always be  $\log(n)$

# Heaps In Python

The screenshot shows a web browser window displaying the Python documentation for the `heapq` module. The title bar reads "8.4. heapq — Heap queue algorithm". The address bar shows the URL <https://docs.python.org/2/library/heappq.html>. The page content is titled "8.4. heapq — Heap queue algorithm". It includes a "Table Of Contents" sidebar with sections like "8.4. heapq — Heap queue algorithm", "8.4.1. Basic Examples", "8.4.2. Priority Queue Implementation Notes", and "8.4.3. Theory". Other links in the sidebar include "Previous topic" (collections), "Next topic" (bisect), "This Page", "Report a Bug", "Show Source", and "Quick search". The main content area describes the heap queue algorithm, its implementation, and provides examples for functions like `heappush`, `heappop`, `heappushpop`, `heapify`, `hearplace`, `merge`, and `nlargest`.

## 8.4. heapq — Heap queue algorithm

New in version 2.3.

[Source code: Lib/heappq.py](#)

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all `k`, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, `heap[0]`.

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our `pop` method returns the smallest item, not the largest (called a "min heap" in textbooks; a "max heap" is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: `heap[0]` is the smallest item, and `heap.sort()` maintains the heap invariant!

To create a heap, use a list initialized to `[]`, or you can transform a populated list into a heap via function `heapify()`.

The following functions are provided:

`heapq.heappush(heap, item)`  
Push the value `item` onto the `heap`, maintaining the heap invariant.

`heapq.heappop(heap)`  
Pop and return the smallest item from the `heap`, maintaining the heap invariant. If the heap is empty, `IndexError` is raised. To access the smallest item without popping it, use `heap[0]`.

`heapq.heappushpop(heap, item)`  
Push `item` on the heap, then pop and return the smallest item from the `heap`. The combined action runs more efficiently than `heappush()` followed by a separate call to `heappop()`.

*New in version 2.6.*

`heapq.heapify(x)`  
Transform list `x` into a heap, in-place, in linear time.

`heapq.hearplace(heap, item)`  
Pop and return the smallest item from the `heap`, and also push the new `item`. The heap size doesn't change. If the heap is empty, `IndexError` is raised.

This one step operation is more efficient than a `heappop()` followed by `heappush()` and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with `item`.

The value returned may be larger than the `item` added. If that isn't desired, consider using `heappushpop()` instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

The module also offers three general purpose functions based on heaps.

`heapq.merge(*iterables)`  
Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an `iterator` over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

*New in version 2.6.*

`heapq.nlargest(n, iterable[, key])`  
Return a list with the `n` largest elements from the data source defined by `iterable`. `key`, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable.

# Heap-based Plane-Sweep

```
## record the delta encoded depth using a plane sweep
deltacovplane = []

## use a list to record the end positions of the elements currently in plane
planeheap = []

## BEGIN SWEEP (note change to index based so can peek ahead)
for rr in xrange(len(reads)):
    r = reads[rr]
    startpos = r[1]
    endpos   = r[2]

    ## clear out any positions from the plane that we have already moved past
    while (len(planeheap) > 0):

        if (planeheap[0] <= startpos):
            ## the coverage steps down, extract it from the front of the list
            ## oldend = planelist.pop(0)
            oldend = heapq.heappop(planeheap)

            nextend = -1
            if (len(planeheap) > 0):
                nextend = planeheap[0]

            ## only record this transition if it is not the same as a start pos
            ## and only if not the same as the next end point
            if ((oldend != startpos) and (oldend != nextend)):
                deltaxcovplane.append((oldend, len(planeheap)))
            else:
                break

        ## Now insert the current endpos into the correct position into the list
        heapq.heappush(planeheap, endpos)

        ## Finally record that the coverage has increased
        ## But make sure the current read does not start at the same position as the next
        if ((rr == len(reads)-1) or (startpos != reads[rr+1][1])):
            deltaxcovplane.append((startpos, len(planeheap)))

        ## if it is at the same place, it will get reported in the next cycle

## Flush any remaining end positions
while (len(planeheap) > 0):
    ##oldend = planelist.pop(0)
    oldend = heapq.heappop(planeheap)
    deltaxcovplane.append((oldend, len(planeheap)))
```

Heaps in python are built from regular lists

planeheap[0] is min

heapq.heappop()  
removes from heap

heapq.heappush() adds to heap

Beginning heap-based plane sweep over 1873 reads

Heap-Plane sweep found 3698 steps, saving 99.62% of the space in 14.26 ms (311.08 speedup)!