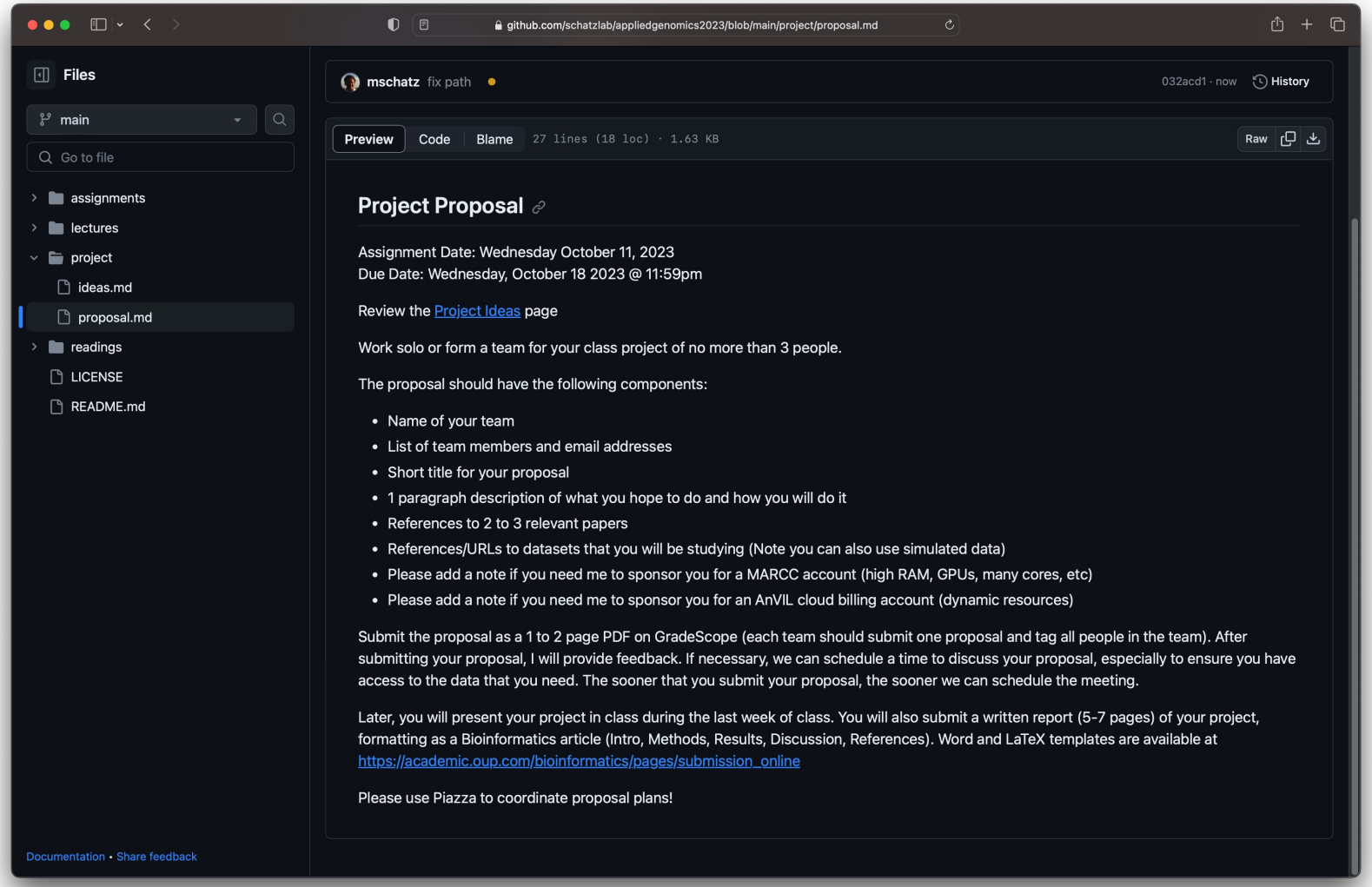
Intro to Annotation

Michael Schatz

October 16, 2023

Lecture 14. Applied Comparative Genomics

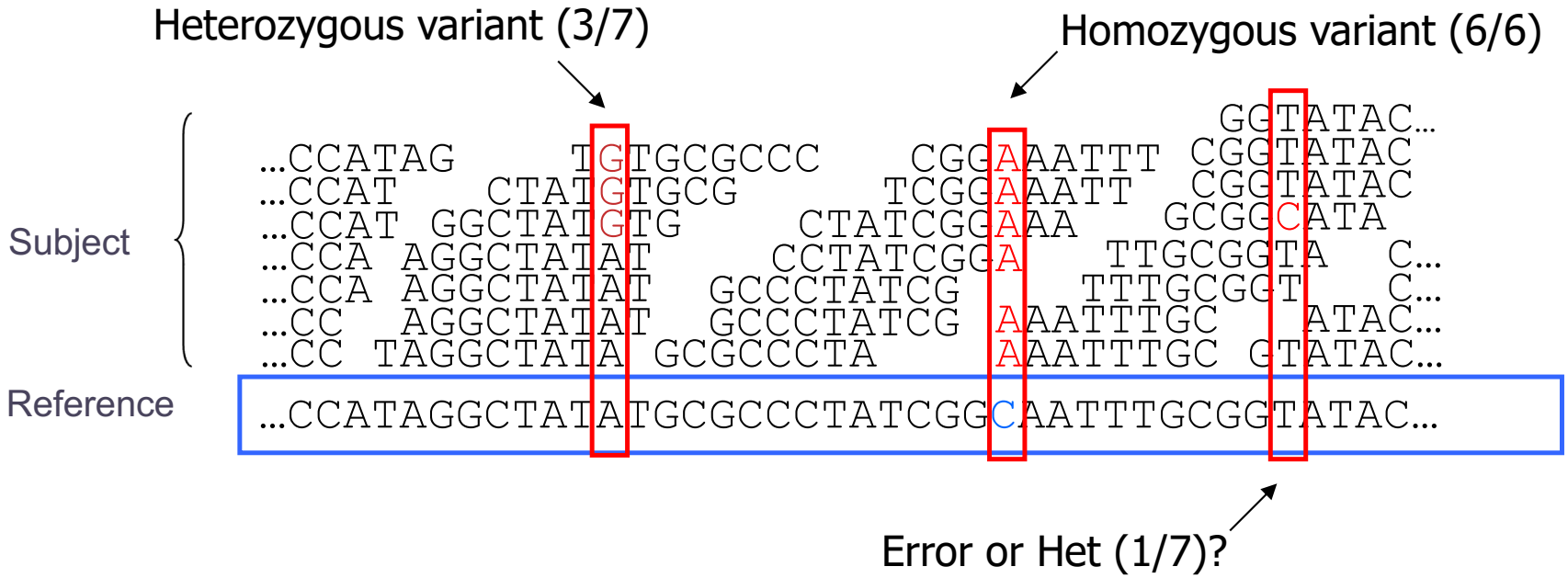
Project Proposal
Due Wednesday Oct 18 by 11:59pm



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

Check Piazza for questions!

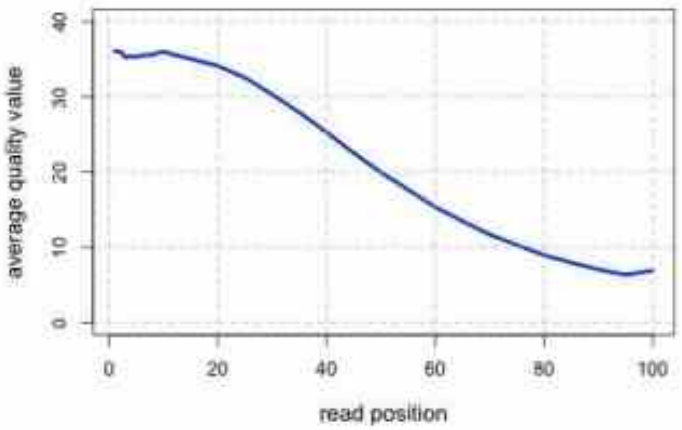
Genotyping Theory



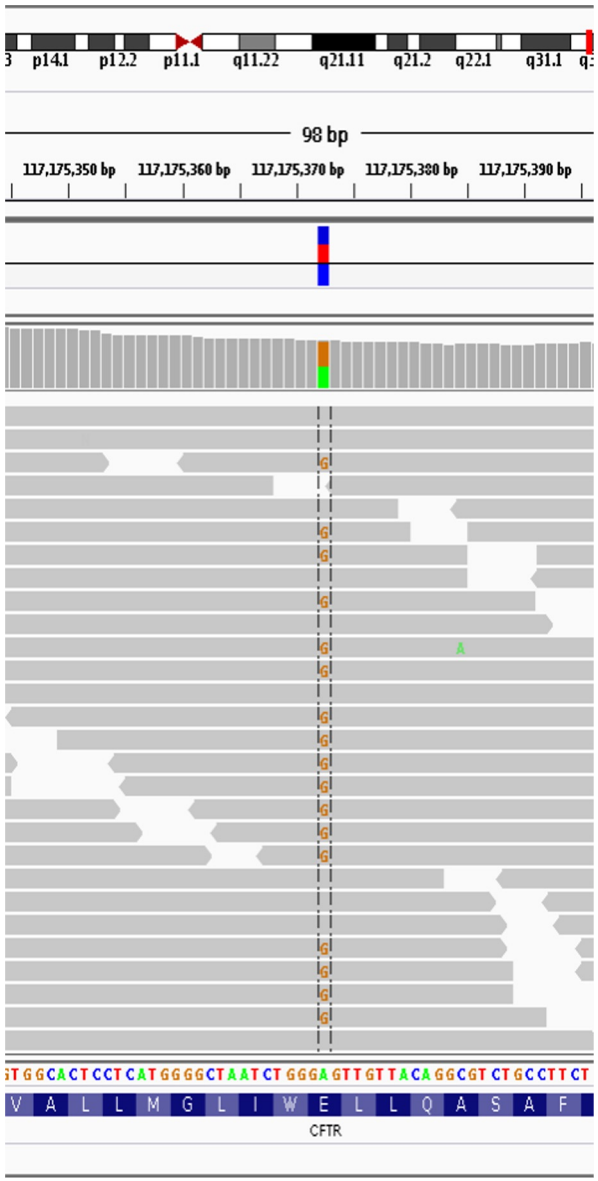
- If there were no sequencing errors, identifying SNPs would be very easy: any time a read disagrees with the reference, it must be a variant!

- 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



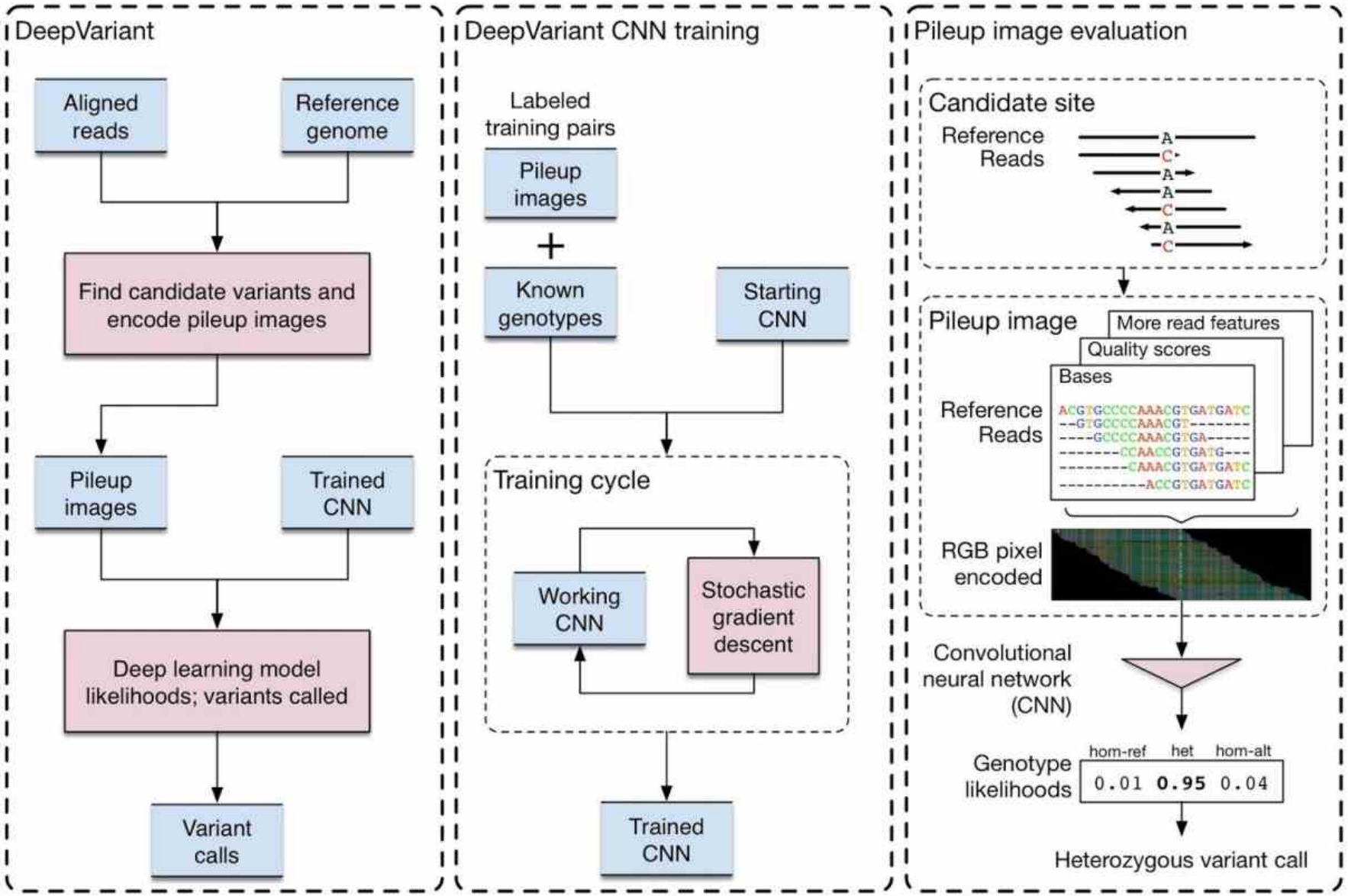
What information is needed to decide if a variant exists?



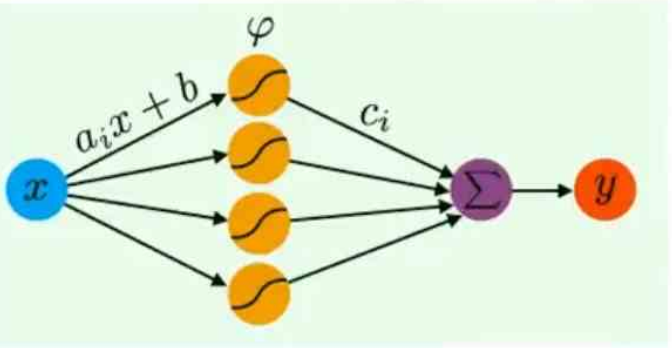
- Depth of coverage at the locus
- Bases observed at the locus
- The base qualities of each allele
- The strand composition
- Mapping qualities
- Proper pairs?
- Expected polymorphism rate

https://genomebiology.biomedcentral.com/articles/10.1186/gb-2010-11-10-r99

DeepVariant



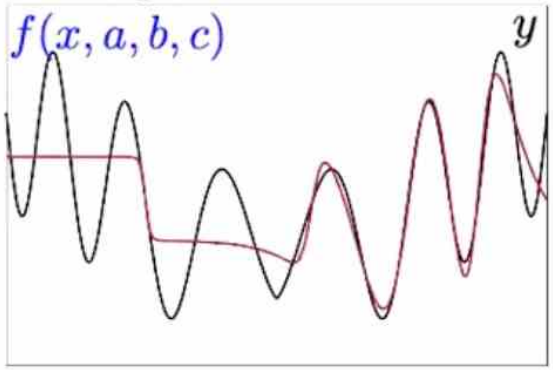
Creating a universal SNP and small indel variant caller with deep neural networks
Poplin et al. (2018) Nature Biotechnology. https://www.nature.com/articles/nbt.4235



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



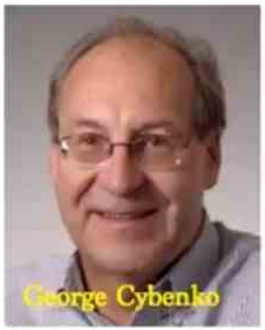
$p = 20$ neurons

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

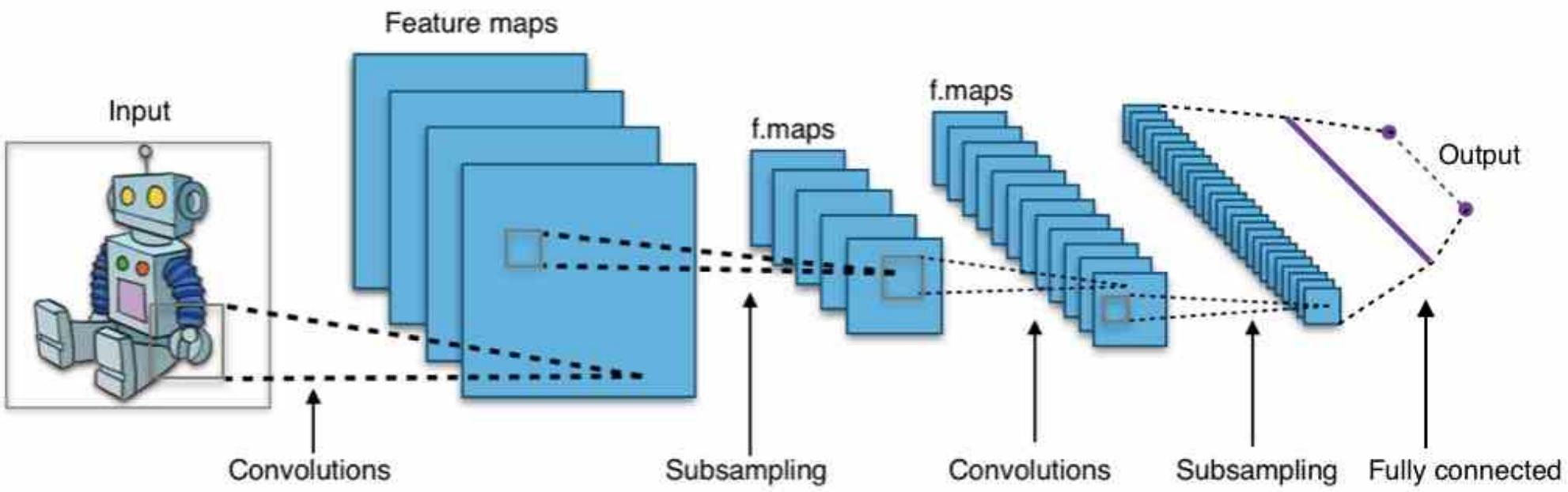
Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate 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.

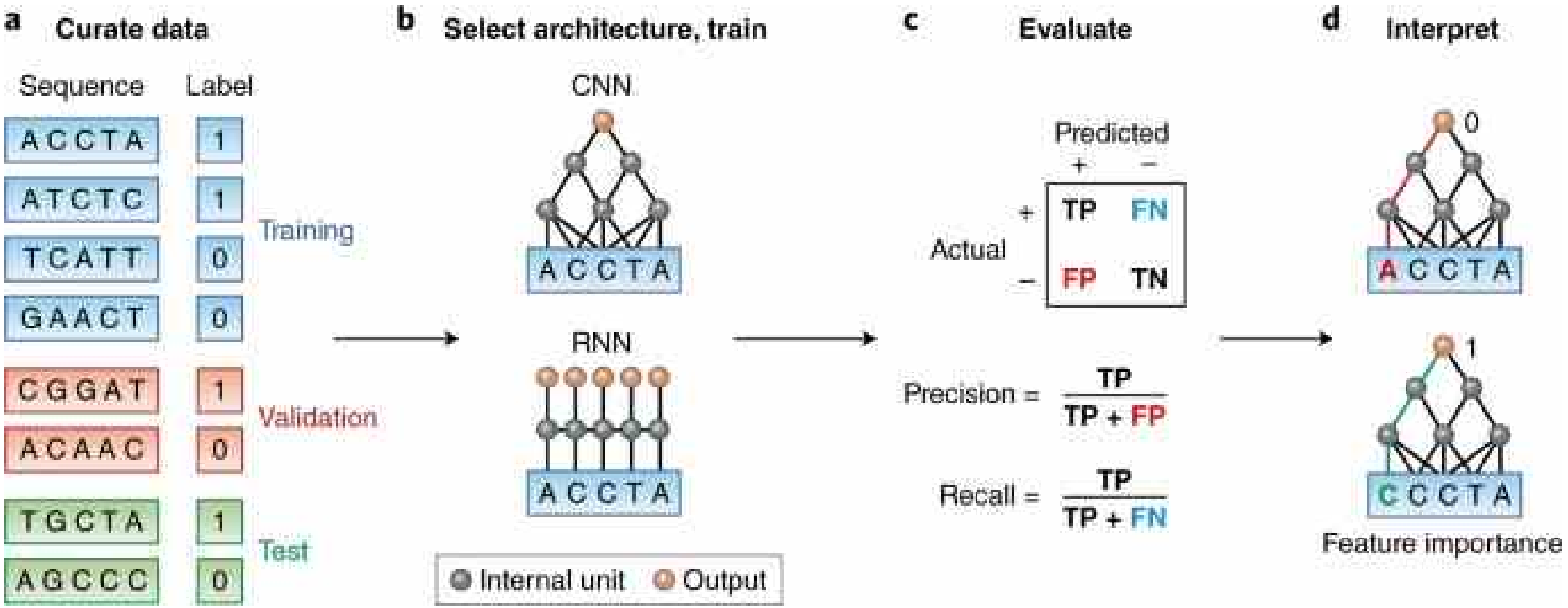


Approximation by superpositions of a sigmoidal function

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



https://training.galaxyproject.org/training-material/topics/statistics/tutorials/CNN/tutorial.html



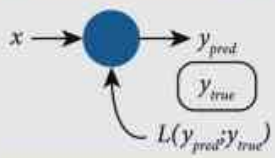
A primer on deep learning in genomics
Zou et al (2019) Nature Genetics. https://www.nature.com/articles/s41588-018-0295-5

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

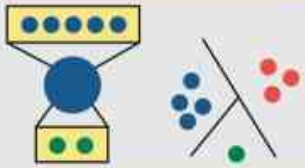
Machine learning scenarios

Supervised learning



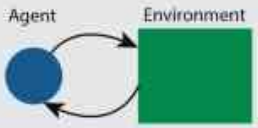
ML algorithm learns by comparing predicted and actual values

Unsupervised learning



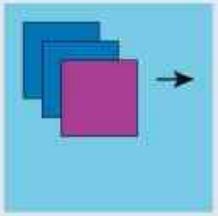
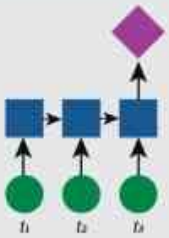
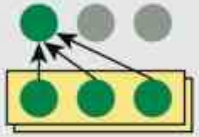
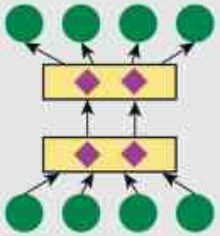
ML algorithm learns without labeled data (e.g. clustering, embedding)

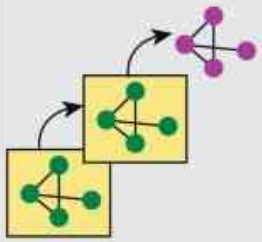
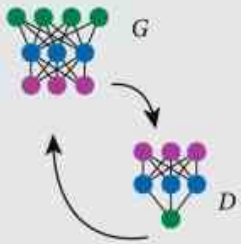
Reinforcement learning



Agent (ML algorithm) learns by interacting with an environment

Key deep learning architectures

	Convolutional NN (CNN)	Recurent NN (RNN)	Transformer	Autoencoder (AE)
Goal	Perform inference on data with local features	Perform inference on temporal data	Perform inference on sequential data	Embed high-dimensional data
Key idea	Learn shift-invariant filters	Learn temporal correlations via recurrent structure	Learn context based correlations via attention mechanism	Learn low-dimensional embedding of data
				

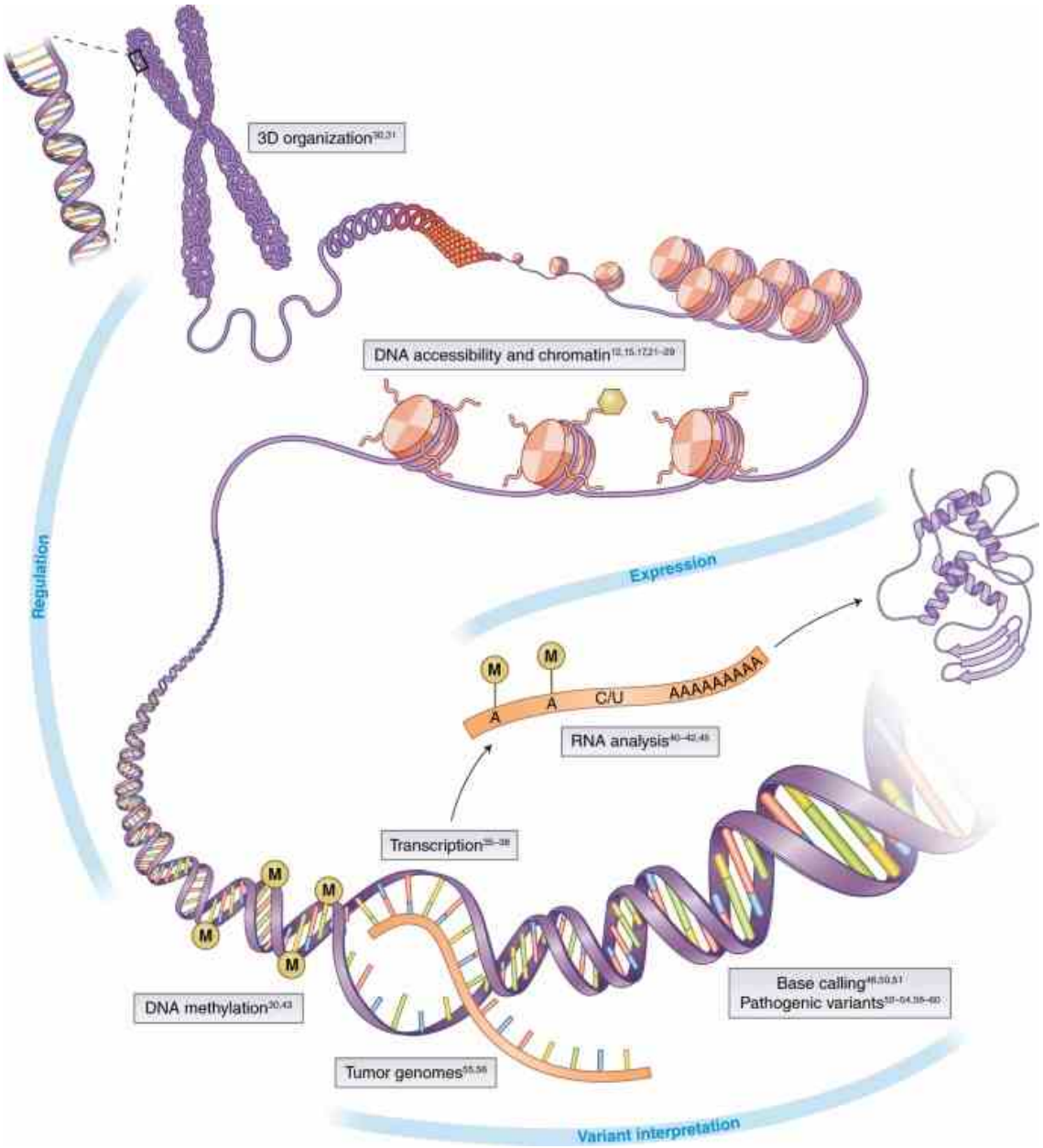
	Graph NN (GNN)	Generative Adversarial Network (GAN)
Goal	Capture graph based dependencies in the data	Generate samples from data distribution
Key idea	Perform message passing between nodes in a layer	Simultanoeusly train generator and discriminator
		

Denoising autoencoders (DAE) 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.

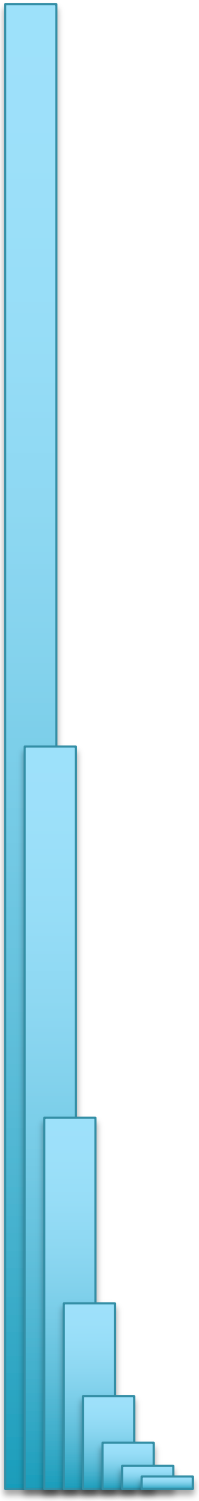
Attention mechanism mimics cognitive attention by learning importance weights for the inputs based on the whole input context (e.g. in a task of translating codons to amino acids attention mechanism will learn to give higher weight to the first two nucleid acids). Attention is the key part of transformer models, but can also be applied in conjunction with other layer types.

Convolutional layers 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.

Graph convolutional network (GCN) 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

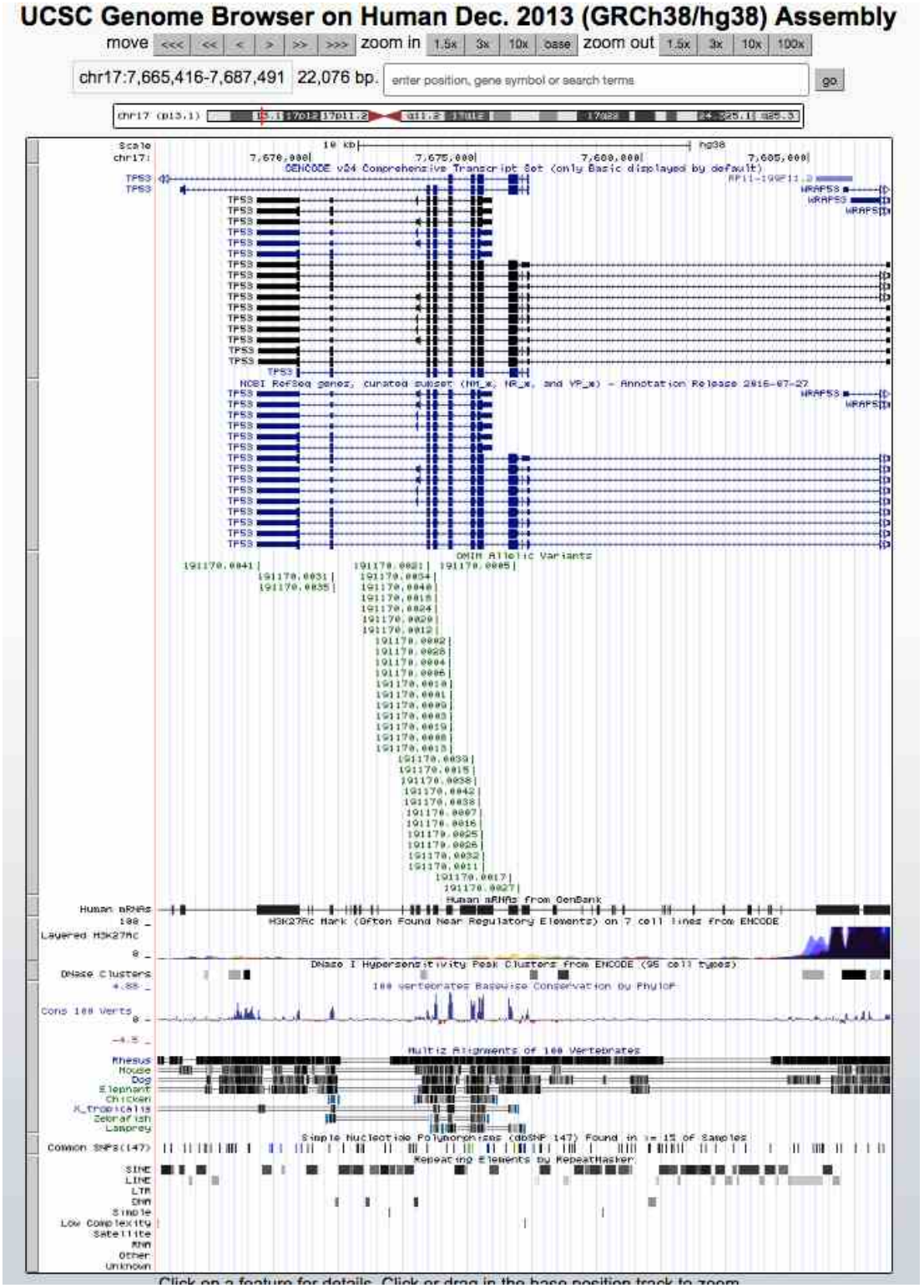
aatgcatgcggctatgctaatgcatgcggctatgctaagctgggatccgatgacaatgcatgcggctatgctaat
gcatgcggctatgcaagctgggatccgatgactatgctaagctgggatccgatgacaatgcatgcggctatgct
aatgaatggtcttgggatttaccttggaatgctaagctgggatccgatgacaatgcatgcggctatgctaatgaa
tggtcttgggatttaccttggaatatgctaatgcatgcggctatgctaagctgggatccgatgacaatgcatgcg
gctatgctaatgcatgcggctatgcaagctgggatccgatgactatgctaagctgcggctatgctaatgcatgcg
gctatgctaagctgggatccgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctgggatcct
gcggctatgctaatgaatggtcttgggatttaccttggaatgctaagctgggatccgatgacaatgcatgcggct
atgctaatgaatggtcttgggatttaccttggaatatgctaatgcatgcggctatgctaagctgggaatgcatgcg
gctatgctaagctgggatccgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctgggatccg
atgactatgctaagctgcggctatgctaatgcatgcggctatgctaagctcatgcggctatgctaagctgggaat
gcatgcggctatgctaagctgggatccgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctg
ggatccgatgactatgctaagctgcggctatgctaatgcatgcggctatgctaagctcggctatgctaatgaatg
gtcttgggatttaccttggaatgctaagctgggatccgatgacaatgcatgcggctatgctaatgaatggtcttgg
gatttaccttggaatatgctaatgcatgcggctatgctaagctgggaatgcatgcggctatgctaagctgggatc
cgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctgggatccgatgactatgctaagctgcg
gctatgctaatgcatgcggctatgctaagctcatgcgg

Goal: Genome Annotations

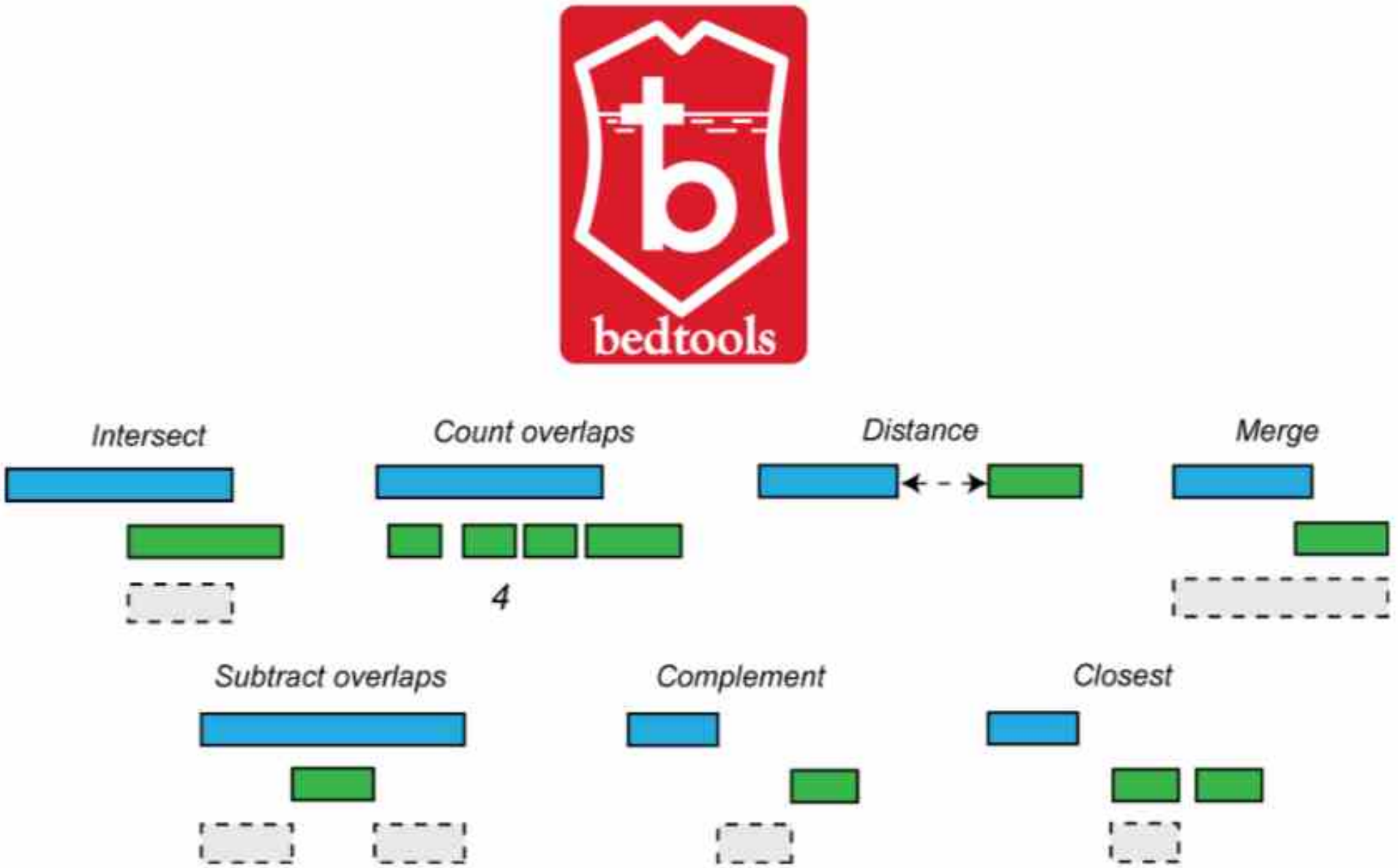
aatgcatgcggctatgctaatgcatgcggctatgctaagctgggatccgatgacaatgcatgcggctatgctaat
gcatgcggctatgcaagctgggatccgatgactatgctaagctgggatccgatgacaatgcatgcggctatgct
aatgaatggtcttgggatttaccttggaatgctaagctgggatccgatgacaatgcatgcggctatgctaatgaa
tggtcttgggatttaccttggaatatgctaatgcatgcggctatgctaagctgggatccgatgacaatgcatgcg
gctatgctaatgcatgcggctatgcaagctgggatccgatgactatgctaagctgcggctatgctaatgcatgcg
gctatgctaagctgggatccgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctgggatcct
gcggctatgctaatgaatggtcttgggatttaccttggaatgctaagctgggatccgatgacaatgcatgcggct
atgctaatgaatggtcttgggatt[Gene!]gctatgctaagctgggaatgcatgcg
gctatgctaagctgggatccgat[redacted]atgcggctatgcaagctgggatccg
atgactatgctaagctgcggctatgctaatgcatgcggctatgctaagctcatgcggctatgctaagctgggaat
gcatgcggctatgctaagctgggatccgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctg
ggatccgatgactatgctaagctgcggctatgctaatgcatgcggctatgctaagctcggctatgctaatgaatg
gtcttgggatttaccttggaatgctaagctgggatccgatgacaatgcatgcggctatgctaatgaatggtcttgg
gatttaccttggaatatgctaatgcatgcggctatgctaagctgggaatgcatgcggctatgctaagctgggatc
cgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctgggatccgatgactatgctaagctgcg
gctatgctaatgcatgcggctatgctaagctcatgcgg

What are genome intervals?

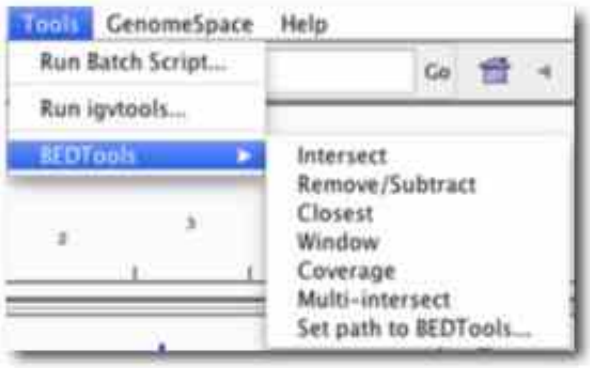
- Genetic variation:
  - SNPs: 1bp
  - Indels: 1-50bp
  - SVs: >50bp
- Genes:
  - exons, introns, UTRs, promoters
- Conservation
- Transposons
- Origins of replication
- TF binding sites
- CpG islands
- Segmental duplications
- Sequence alignments
- Chromatin annotations
- Gene expression data
- …
-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



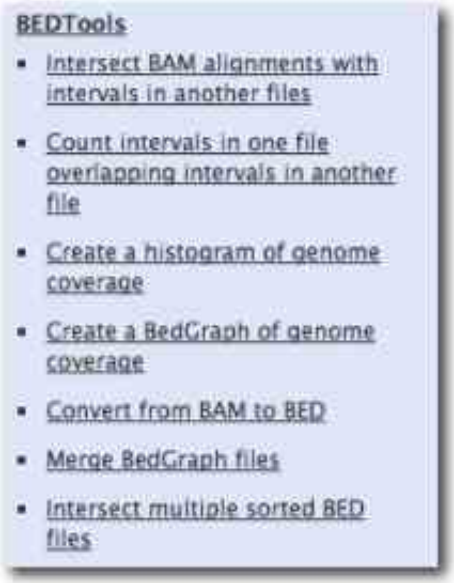
BEDTools to the rescue!



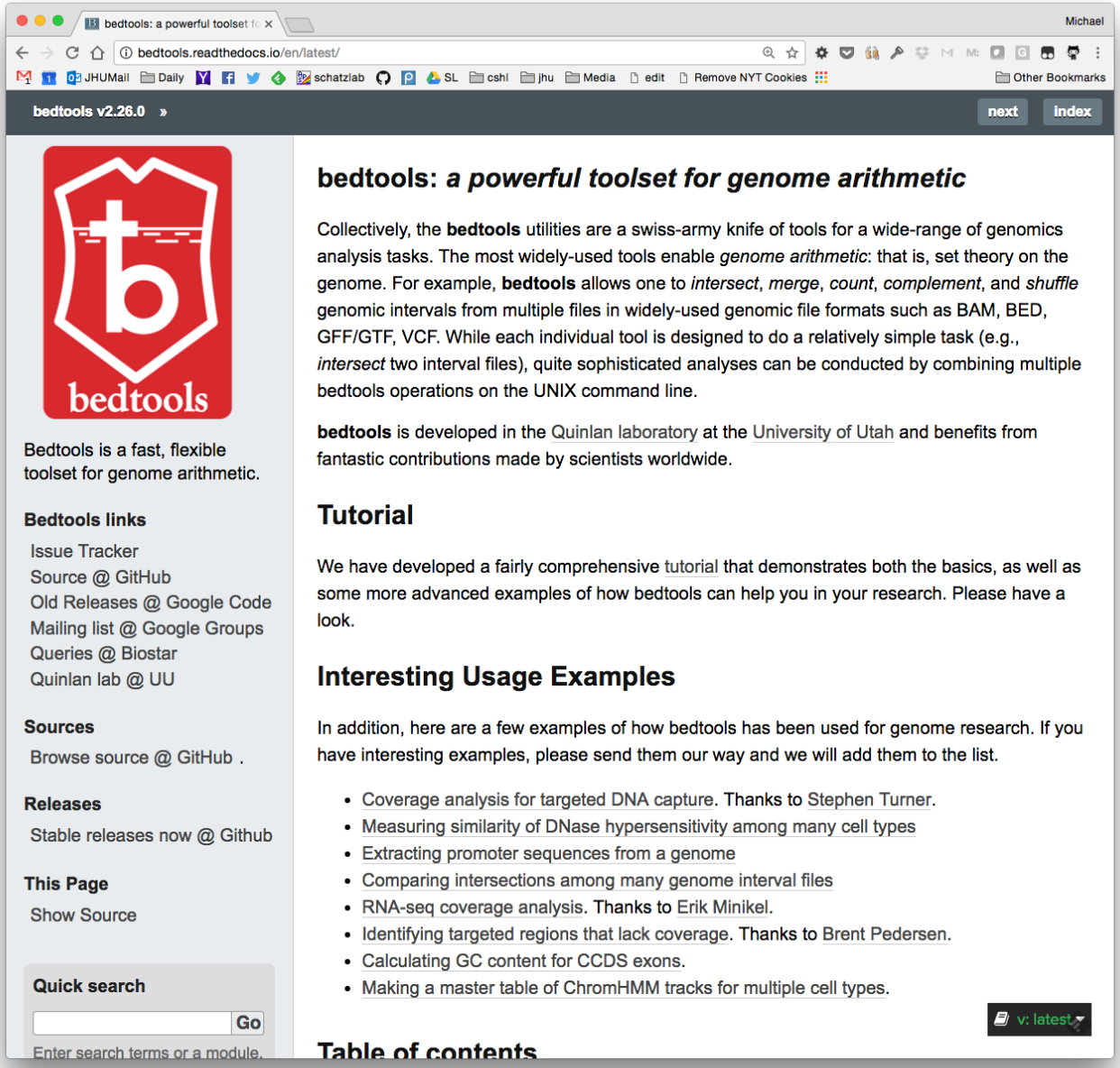
Getting & Using BEDTools



Integrated into **IGV**



In **Galaxy** Toolshed



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:

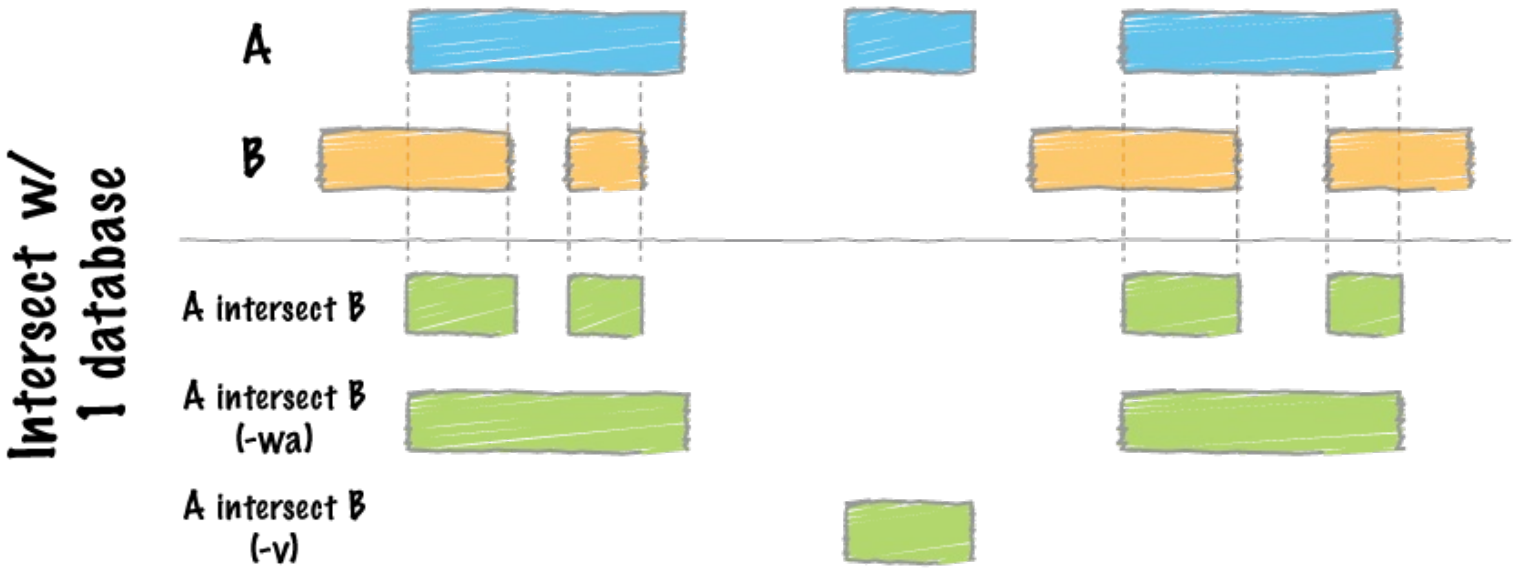
1. chrom — The name of the chromosome (e.g. chr3, chrY, chr2_random) or scaffold (e.g. scaffold10671).
2. chromStart — The starting position of the feature in the chromosome or scaffold. The first base in a sequence is numbered 0.
3. chromEnd — The ending position of the feature in the chromosome or 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



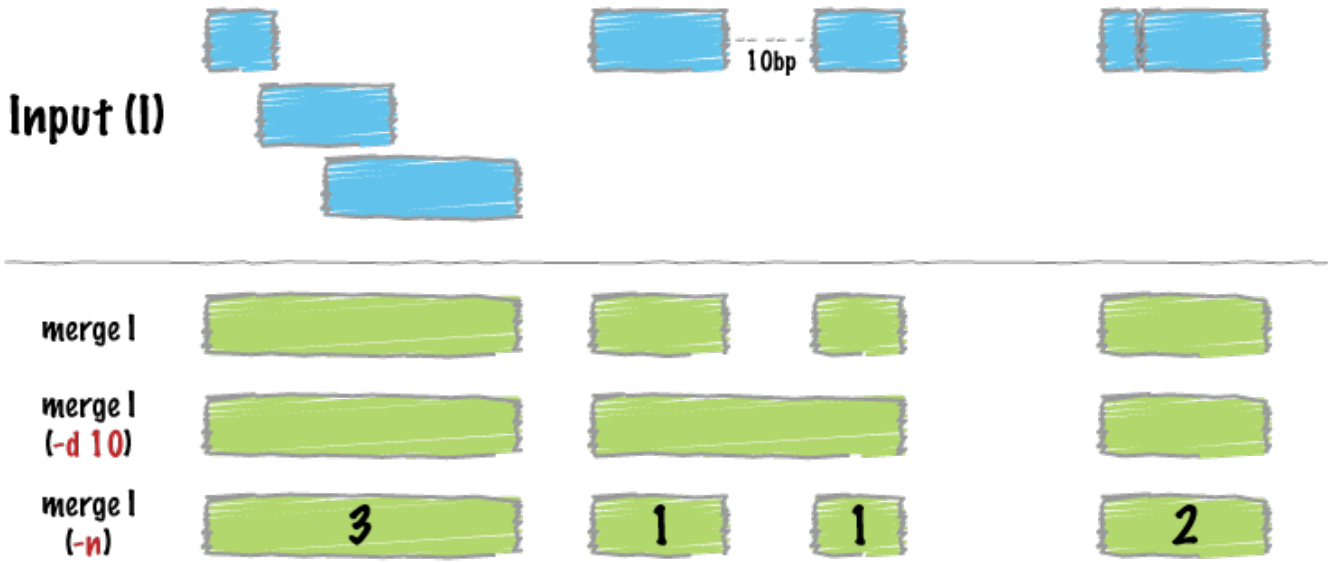
What exons are hit by SVs?

```
$ 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
```

Note input must be sorted!

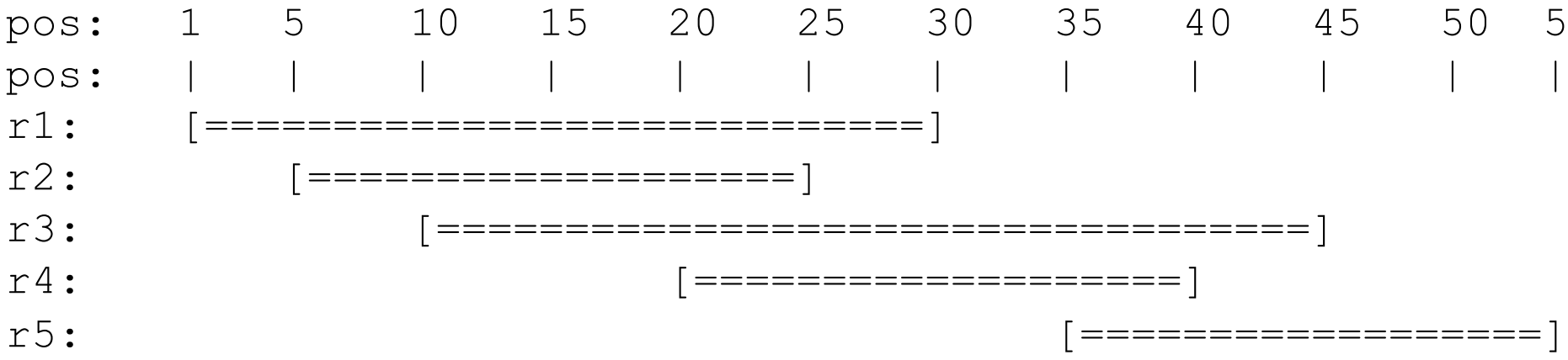
```
sort -k1,1 -k2,2n foo.bed > foo.sort.bed
```

BEDTools commands

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

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

Plane Sweep



The basic algorithm works like this:

- Assume layout is in sorted order by start position (or explicitly sort by start position)

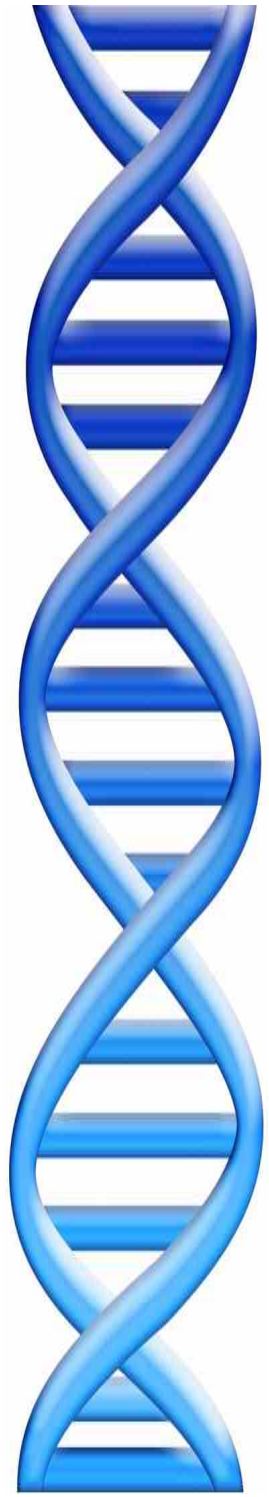
- use a “list” to track how many reads currently intersect the plane keyed by end coord
  - the number of elements in the list corresponds to the current depth

- 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!

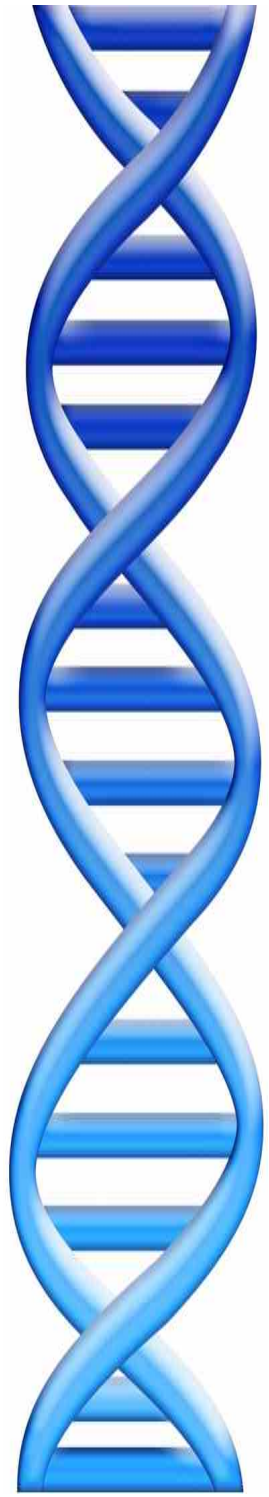
Goal: Genome Annotations

aatgcatgcggctatgctaatgcatgcggctatgctaagctgggatccgatgacaatgcatgcggctatgctaat
gcatgcggctatgcaagctgggatccgatgactatgctaagctgggatccgatgacaatgcatgcggctatgct
aatgaatggtcttgggatttaccttggaatgctaagctgggatccgatgacaatgcatgcggctatgctaatgaa
tggtcttgggatttaccttggaatatgctaatgcatgcggctatgctaagctgggatccgatgacaatgcatgcg
gctatgctaatgcatgcggctatgcaagctgggatccgatgactatgctaagctgcggctatgctaatgcatgcg
gctatgctaagctgggatccgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctgggatcct
gcggctatgctaatgaatggtcttgggatttaccttggaatgctaagctgggatccgatgacaatgcatgcggct
atgctaatgaatggtcttgggatt[Gene!]gctatgctaagctgggaatgcatgcg
gctatgctaagctgggatccgat[redacted]atgcggctatgcaagctgggatccg
atgactatgctaagctgcggctatgctaatgcatgcggctatgctaagctcatgcggctatgctaagctgggaat
gcatgcggctatgctaagctgggatccgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctg
ggatccgatgactatgctaagctgcggctatgctaatgcatgcggctatgctaagctcggctatgctaatgaatg
gtcttgggatttaccttggaatgctaagctgggatccgatgacaatgcatgcggctatgctaatgaatggtcttgg
gatttaccttggaatatgctaatgcatgcggctatgctaagctgggaatgcatgcggctatgctaagctgggatc
cgatgacaatgcatgcggctatgctaatgcatgcggctatgcaagctgggatccgatgactatgctaagctgcg
gctatgctaatgcatgcggctatgctaagctcatgcgg



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

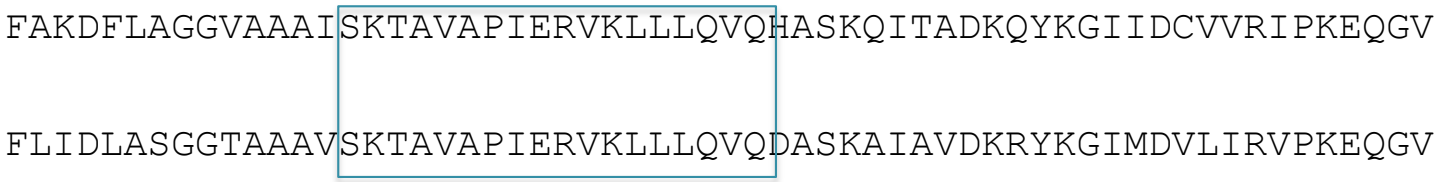
Basic Local Alignment Search Tool

- Rapidly compare a sequence Q to a database to find all sequences in the database with an 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

(Altschul et al. 1990)

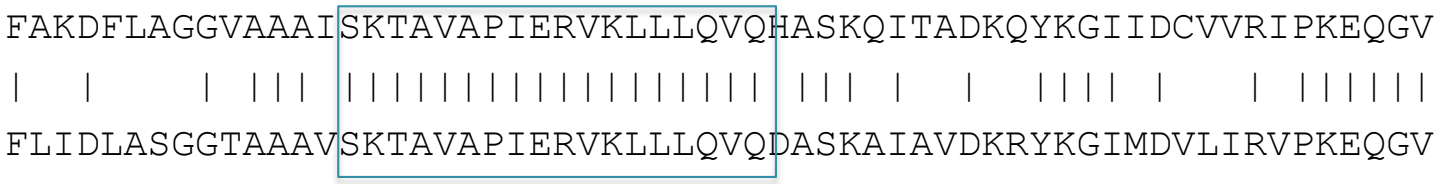
Seed and Extend



- 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



- 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, λ 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   LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF------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 EFTPAVHASLDKFLASVSTVLTSKY 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   LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF------DLSHGSAQV  55
           LS  +   V   WGKV A    +G E L R+F   P T   F  F      D    S  +
Sbjct  3   LSDGEWQLVLNVWGKVEADIPGHGQEVLIRLFKGHPETLEKFDKFKHLKSEDEMKASEDL  62

Query  56  KGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPA  115
           K HG  V  AL   +          +  L+  HA K ++     + +S C++  L +  P
Sbjct  63  KKHGATVLTALGGILKKKGHHEAEIKPLAQSHATKHKIPVKYLEFISECIIQVLQSKHPG  122

Query  116 EFTPAVHASLDKFLASVSTVLTSKYR  141
            +F      +++K L      + S Y+
Sbjct  123 DFGADAQGAMNKALELFRKDMASNYK  148
```

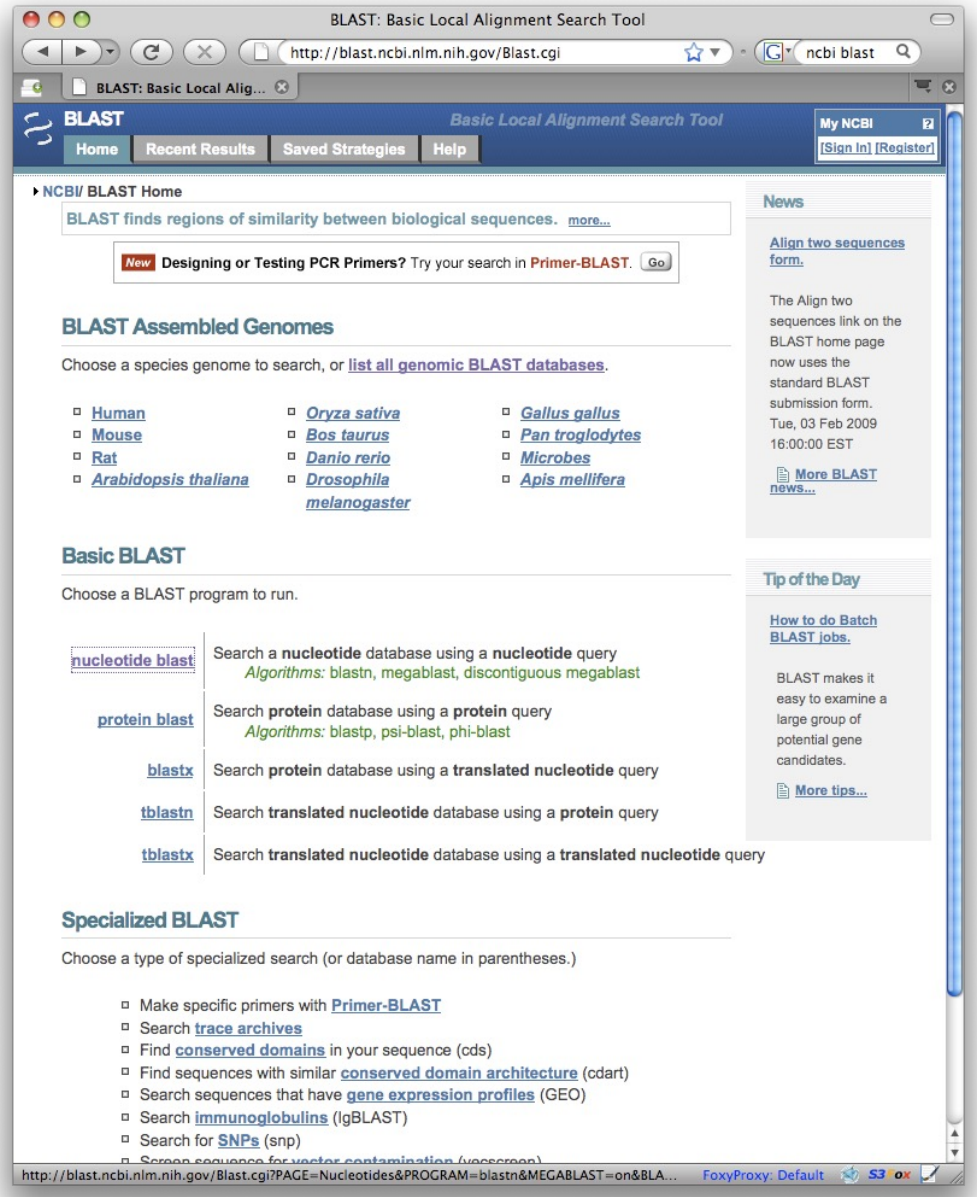
Not similar sequences

```
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  ERMFLSFPTTKTYFPHFDLSHGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAH  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
```

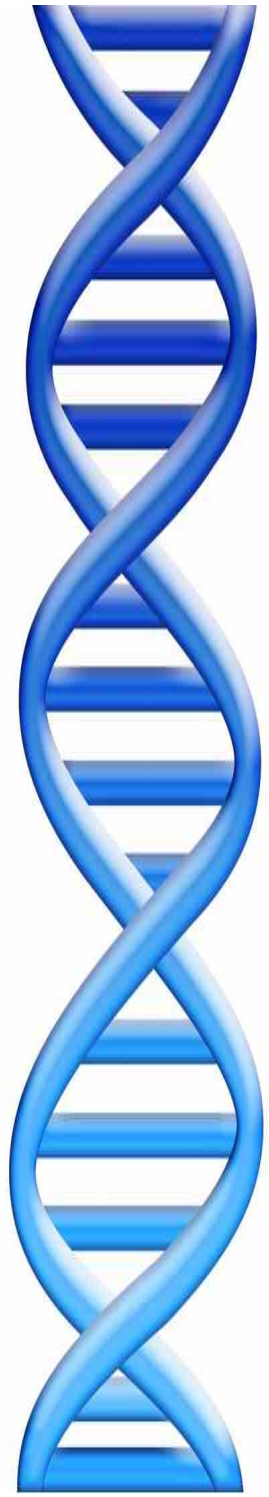
Blast Versions

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



- Nucleotide Databases
  - nr: All Genbank
  - refseq: Reference organisms
  - wgs: All reads
- Protein Databases
  - nr: All non-redundant sequences
  - Refseq: Reference proteins



Outline

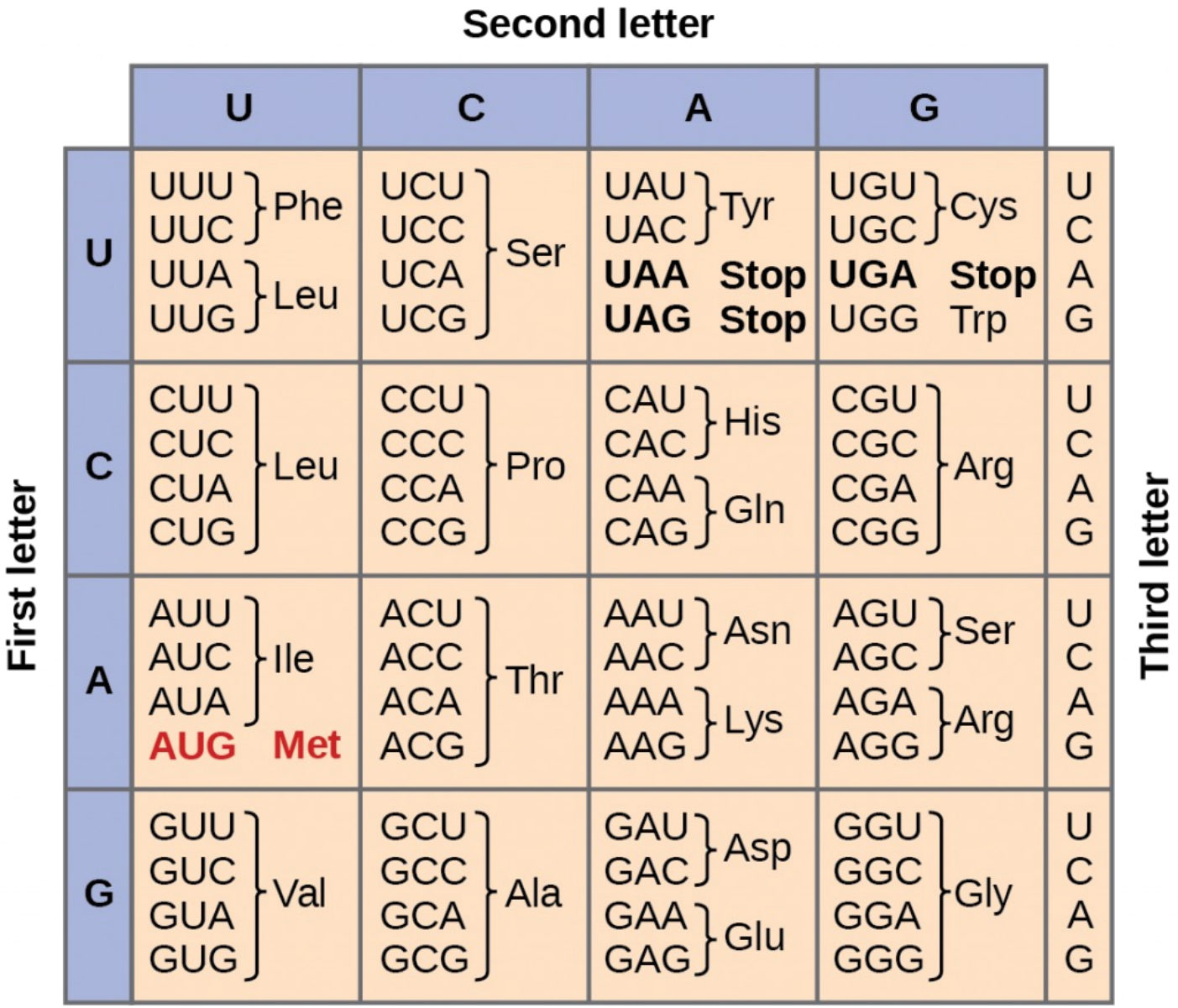
1. Alignment to other genomes
2. Prediction aka "Gene Finding"
3. Experimental & Functional Assays

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



Start:
- AUG

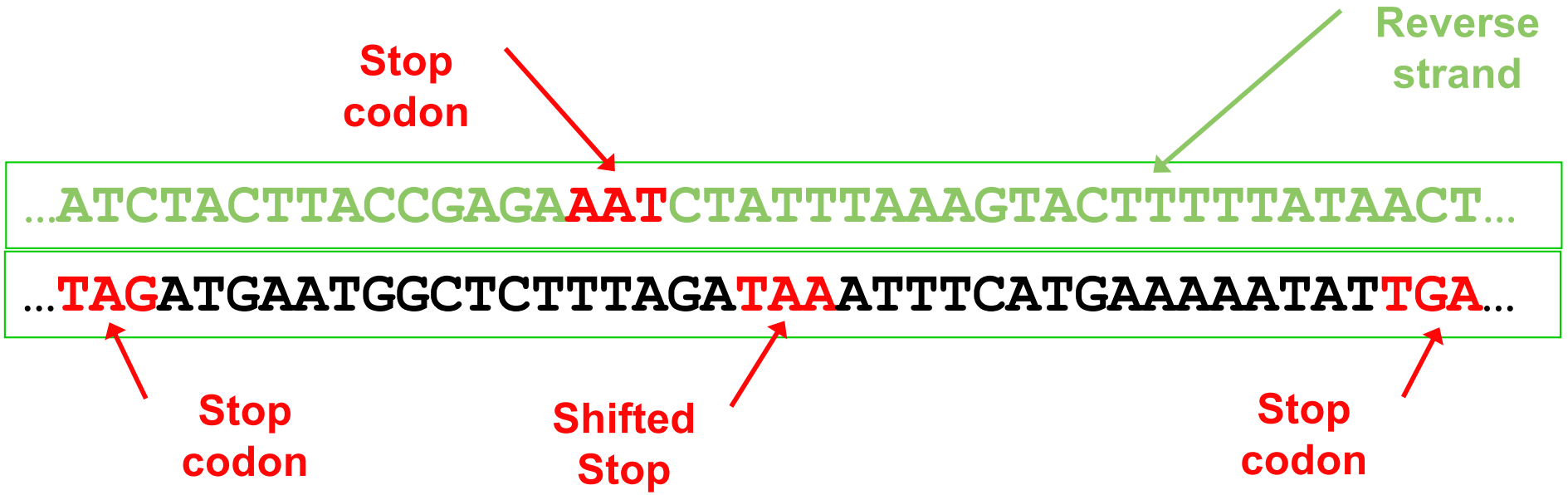
Stop:
- UAA
- UAG
- UGA

Step One

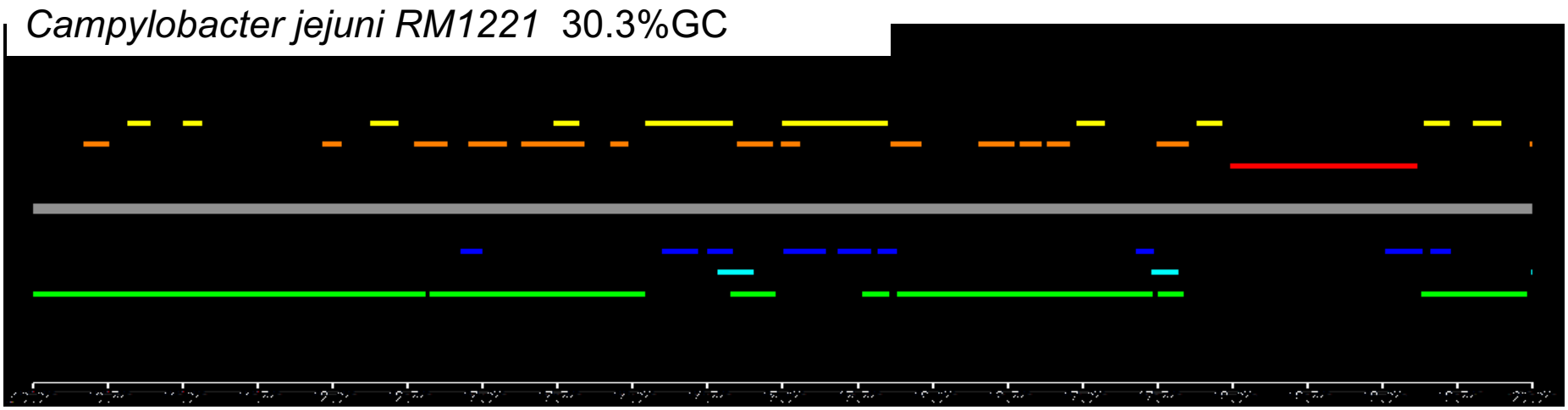
- Find open reading frames (ORFs).

Step One

- Find open reading frames (ORFs).



- But ORFs generally overlap …



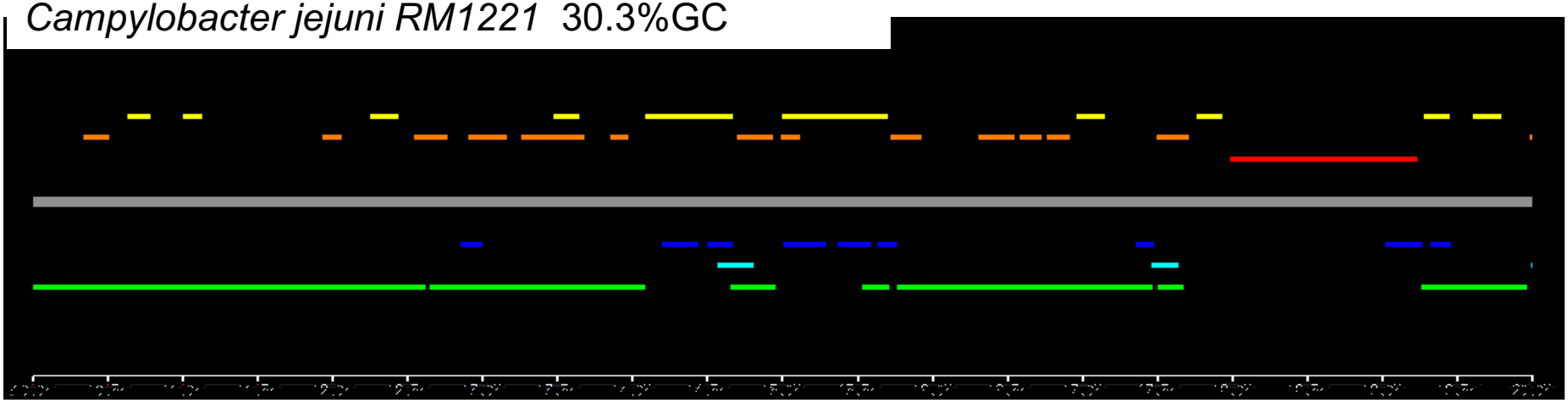
All ORFs longer than 100bp on both strands shown
- color indicates reading frame

Longest ORFs likely to be protein-coding genes

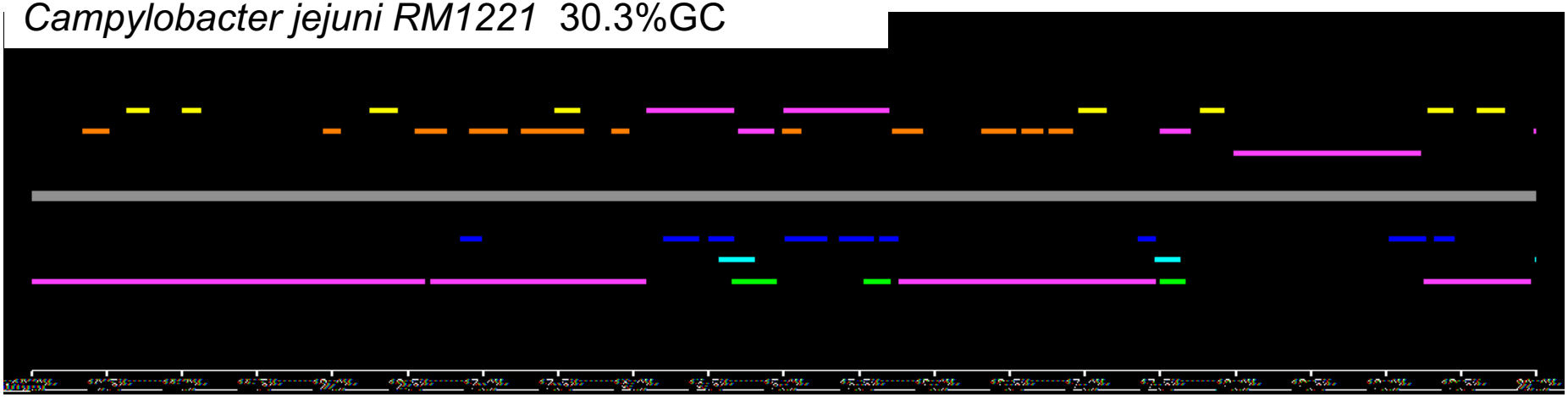
Note the low GC content

All genes are ORFs but not all ORFs are genes

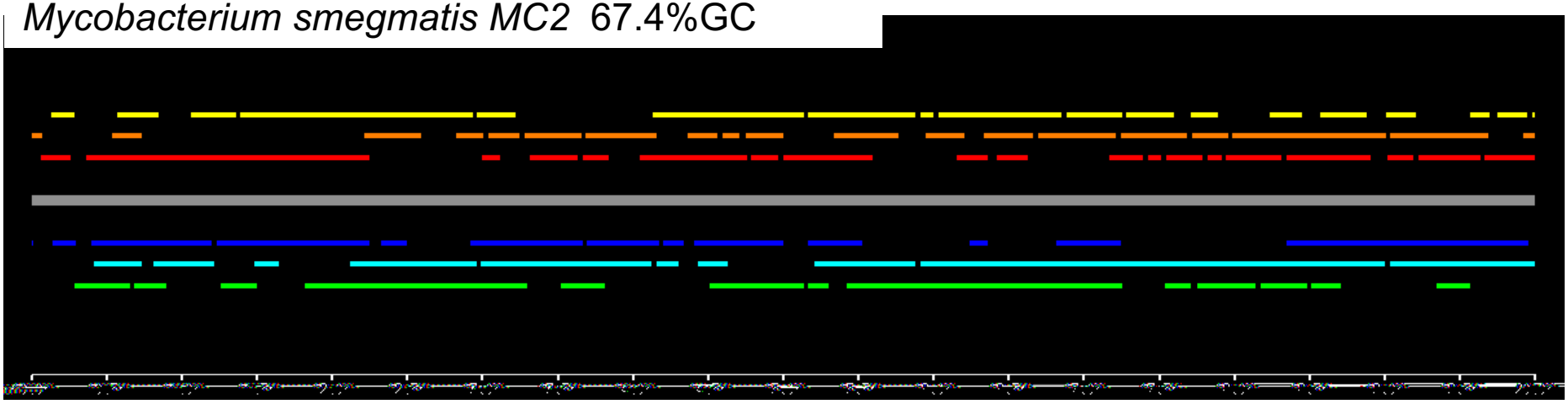
Campylobacter jejuni RM1221 30.3%GC



Campylobacter jejuni RM1221 30.3%GC

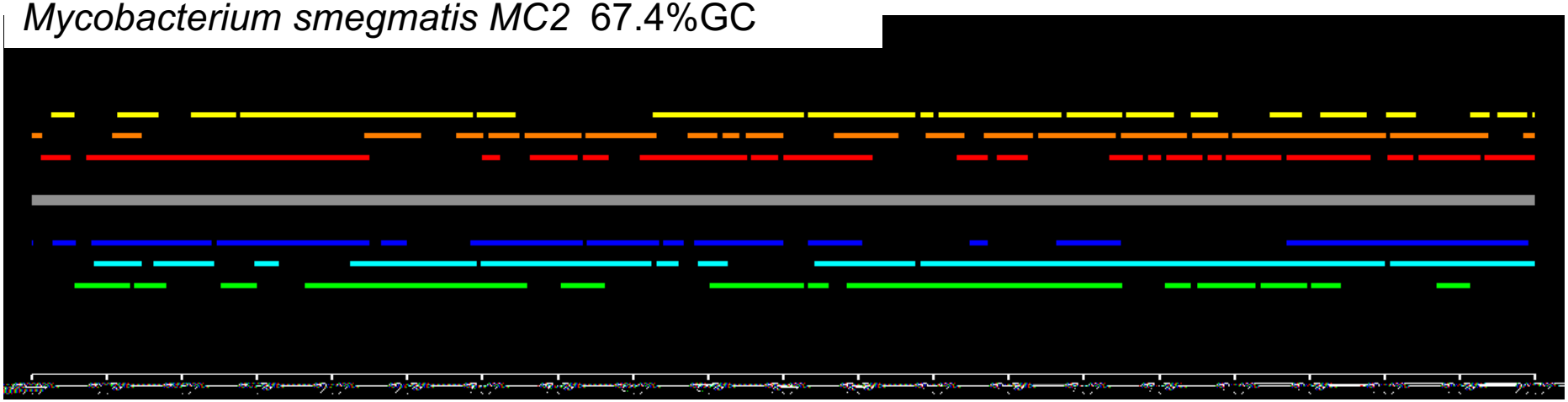


Mycobacterium smegmatis MC2 67.4%GC

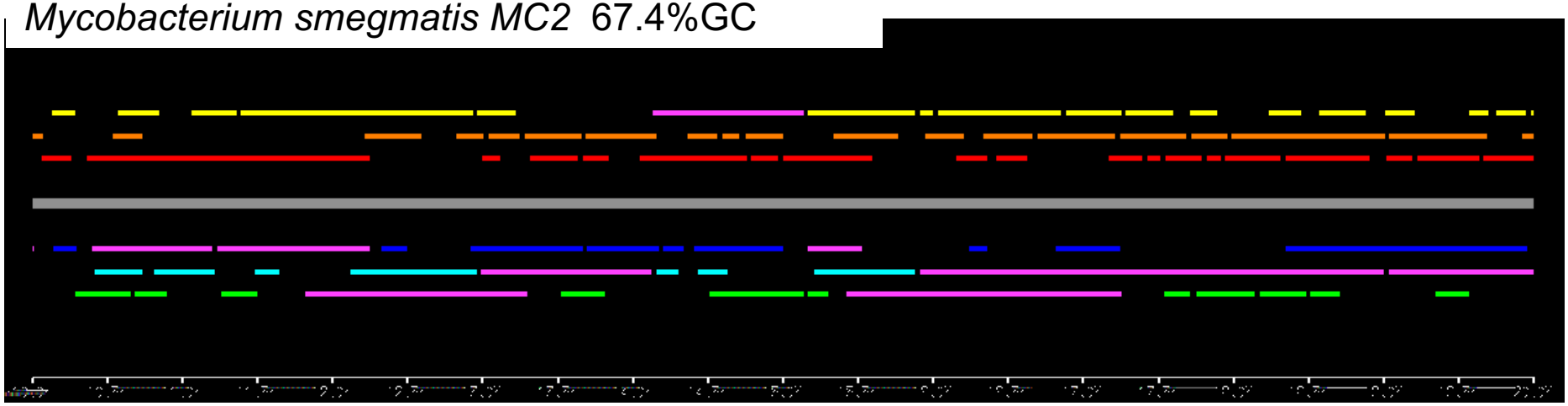


Note what happens in a high-GC genome

Mycobacterium smegmatis MC2 67.4%GC



Mycobacterium smegmatis MC2 67.4%GC



Flipping a Biased Coin

P(heads) = 61/64 (95.4%) P(tails) = 3/64 (4.6%)

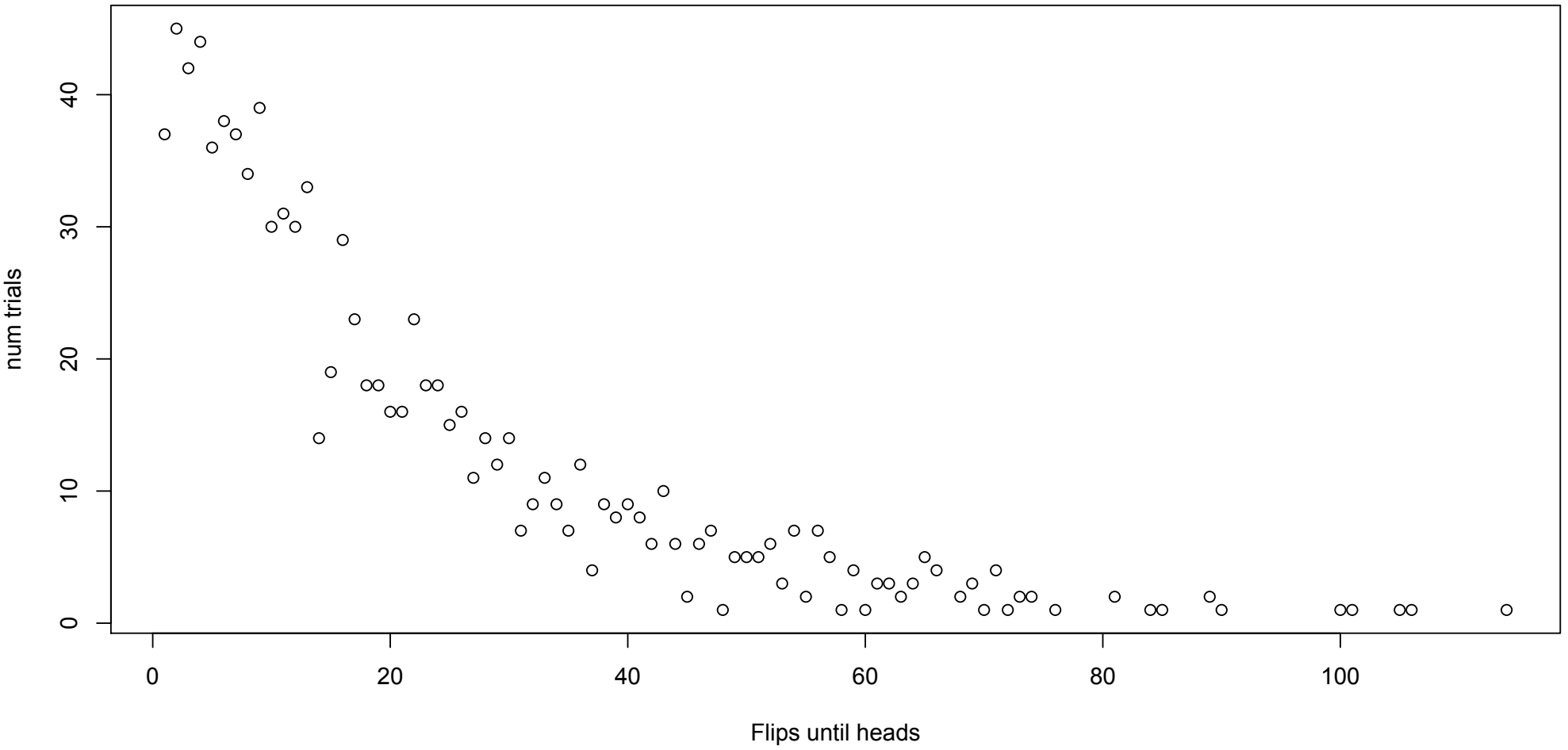
How many flips until my first tail?

```
$ ./coinflip.pl 0.046875 1000
0:	HHHHHHHHHHHHHHT	15
1:	HHHHHHT	7
2:	HHHHHHHHHHHT	12
3:	HHHHHHHHHHHHHHHHHHHHHHHT	24
4:	HT	2
5:	HHHHHHHHHHHHHT	14
6:	HHHHHHHHHT	10
7:	HHHHHHHHHHHHHT	14
8:	HHHHHT	6
9:	HHHHHHHHHHT	11
10:	HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
11:	HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHT	40
12:	HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHT	45
13:	HHHT	4
14:	HHHHHHHHHHHHHHT	15
15:	HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHT	39
16:	HHHHHT	6
17:	HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHT	38
18:	HHHHHHHHHHHHHHHHHHHHHHHHHT	26
19:	HHHHHHHHHHHT	12
```

Flipping a Biased Coin

P(heads) = 61/64 (95.4%) P(tails) = 3/64 (4.6%)

How many flips until my first tail?

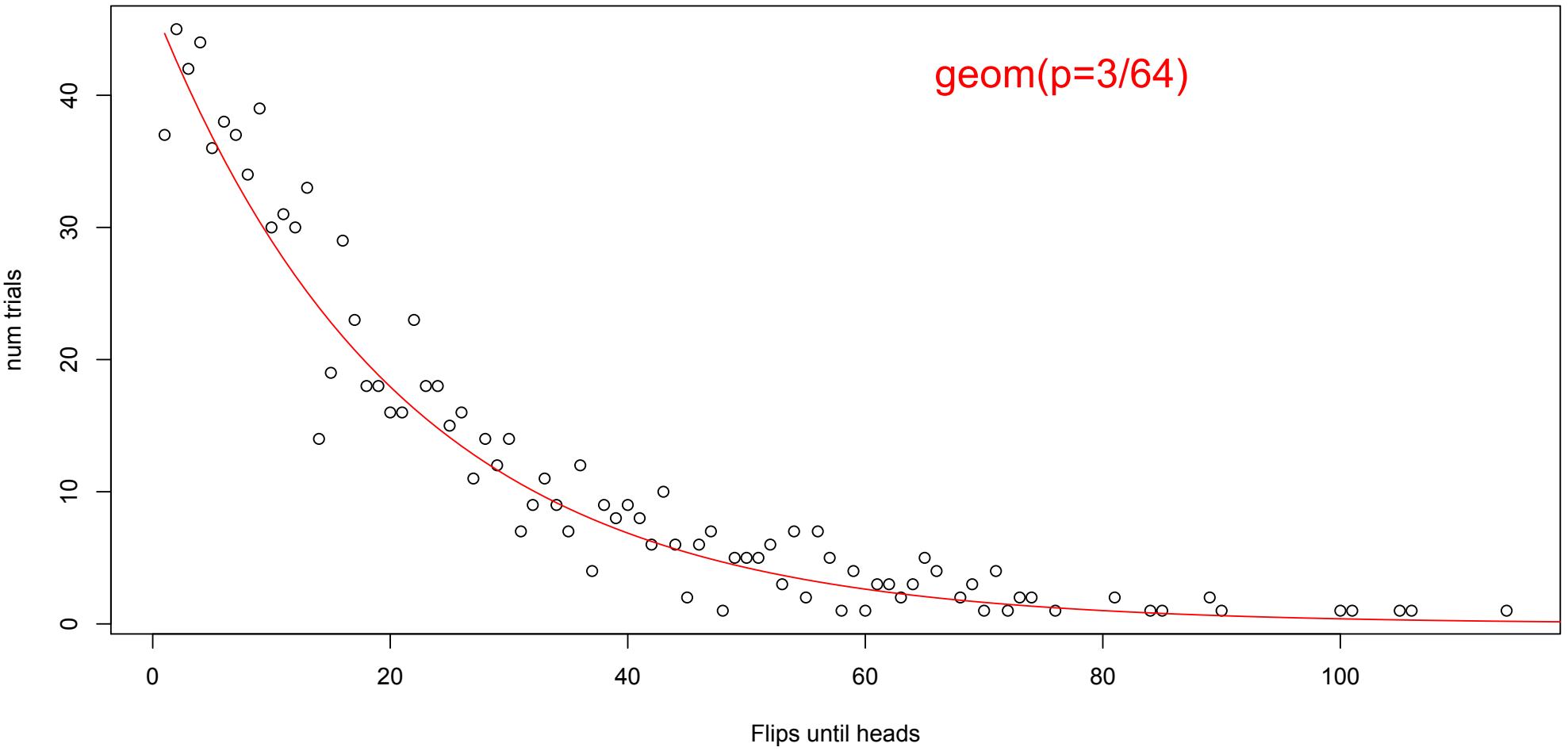


Flipping a Biased Coin

P(heads) = 61/64 (95.4%) P(tails) = 3/64 (4.6%)

How many flips until my first tail?

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

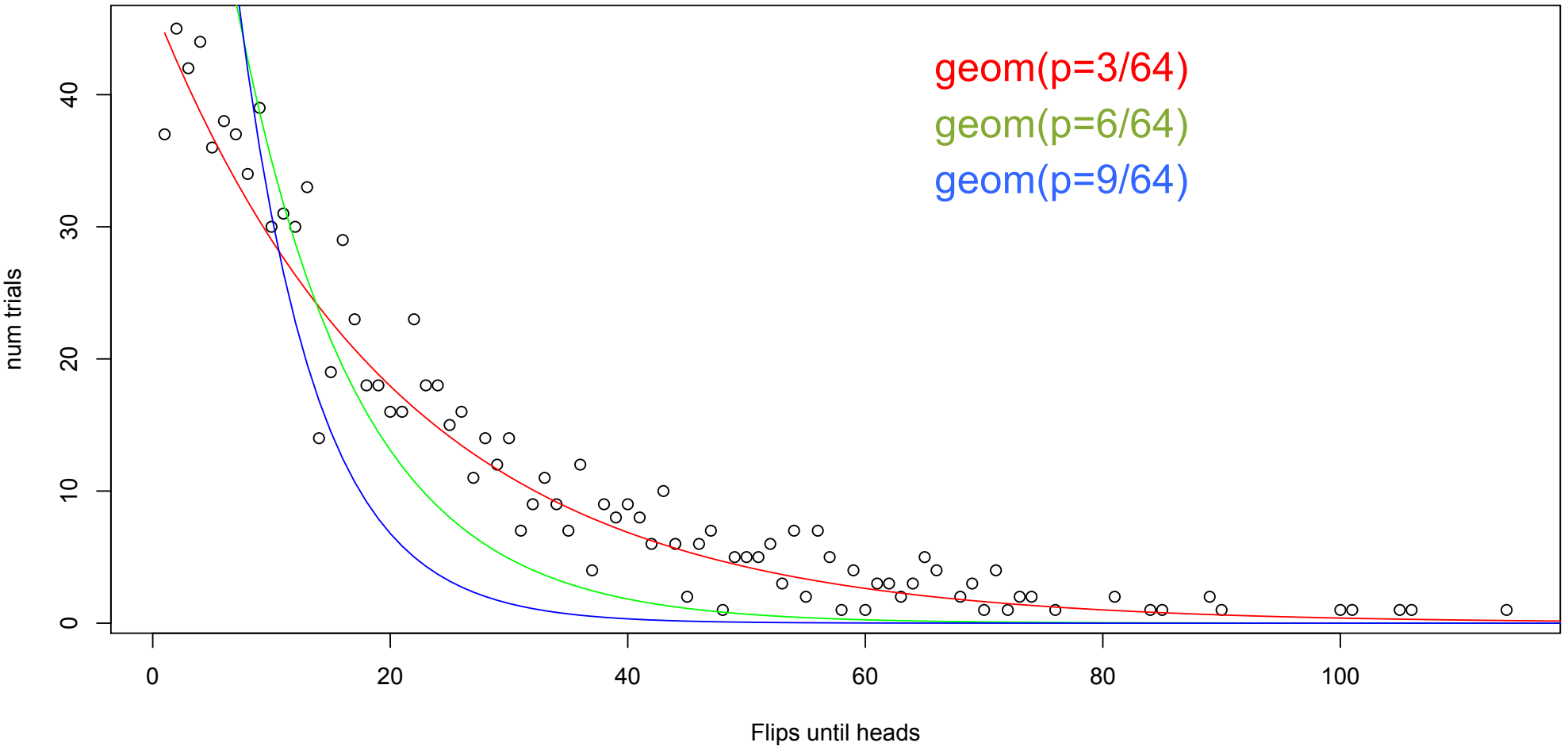


Flipping a Biased Coin

P(heads) = 61/64 (95.4%) P(tails) = 3/64 (4.6%)

How many flips until my first tail?

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

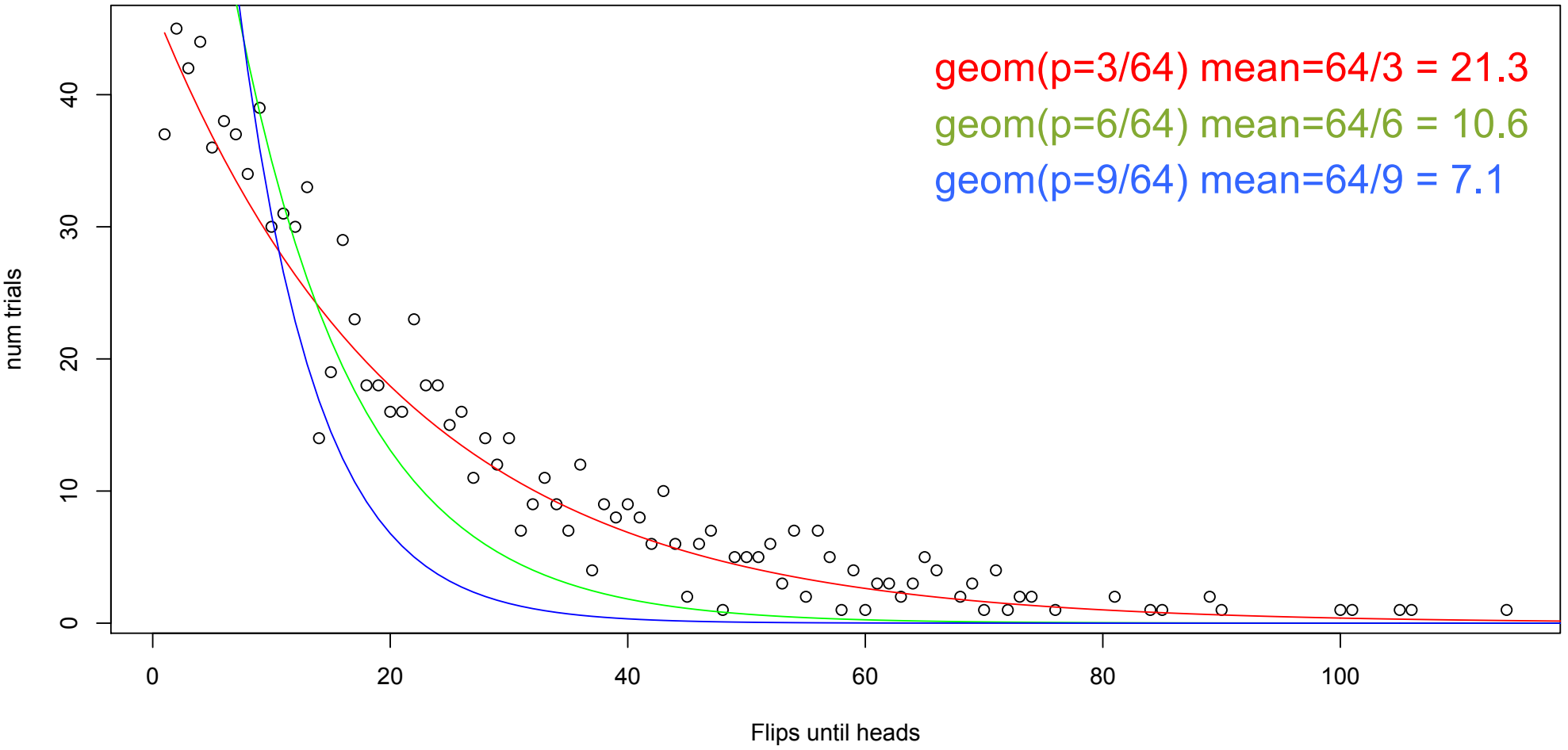


Flipping a Biased Coin

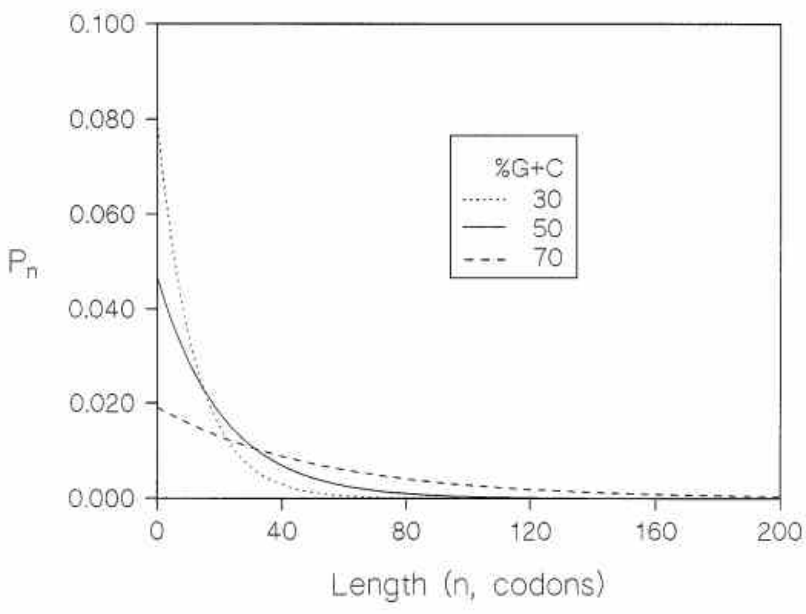
P(heads) = 61/64 (95.4%) P(tails) = 3/64 (4.6%)

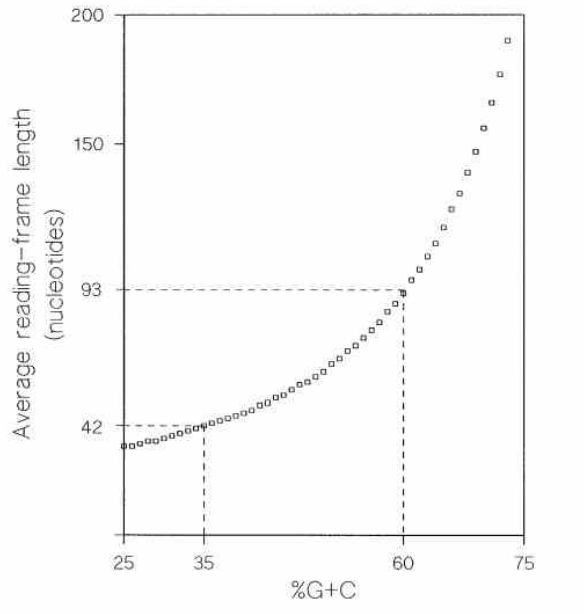
How many flips until my first tail?

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



Stop Codon Frequencies



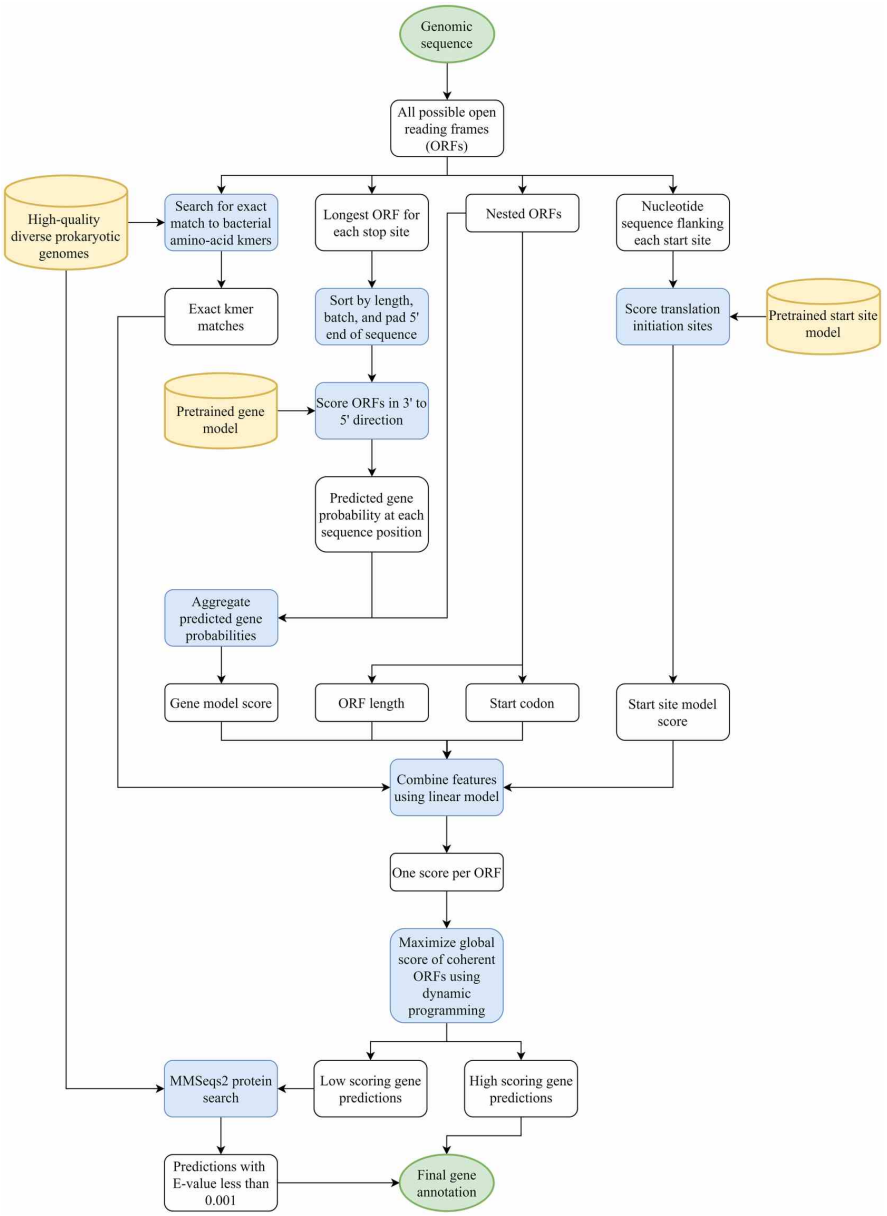


If the sequence is mostly A+T, then likely to form stop codons by chance!

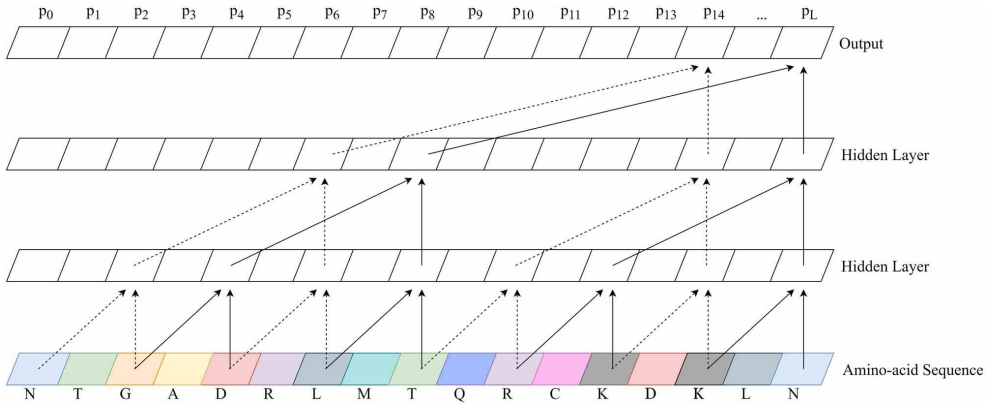
In High A+T (Low G+C):
Frequent stop codons; Short Random ORFs; long ORFs likely to be true genes

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

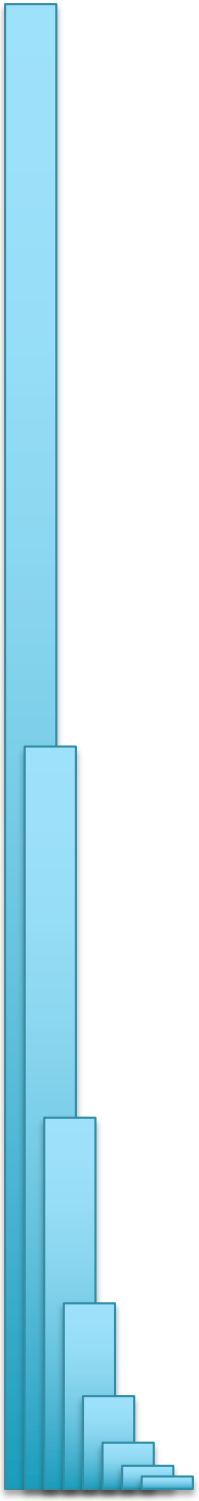


Balrog: A universal protein model for prokaryotic gene prediction

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

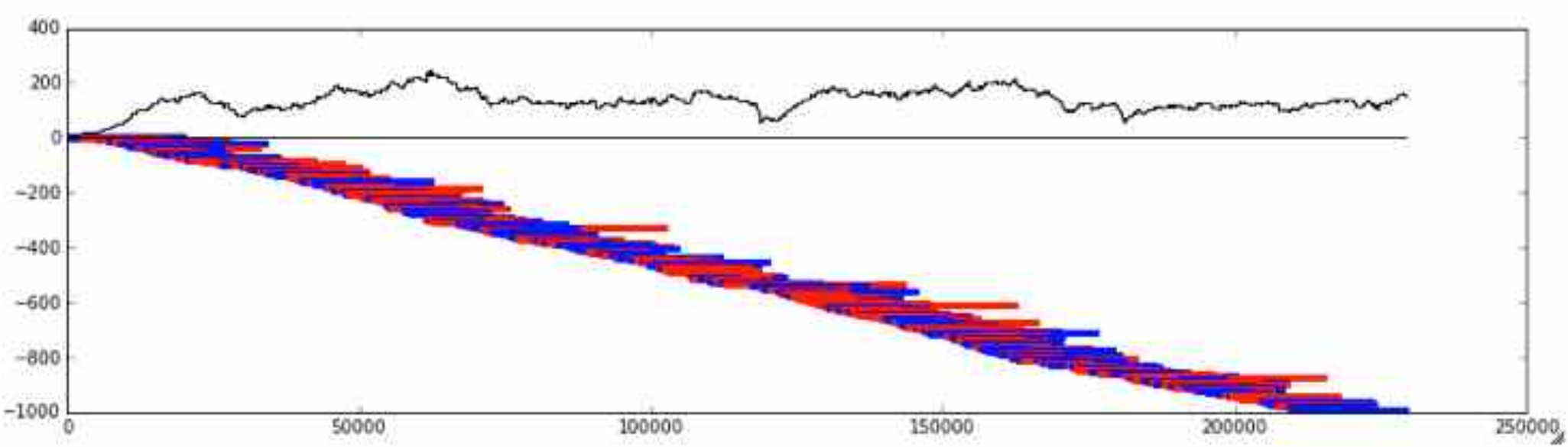
Probabilistic Methods

- 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



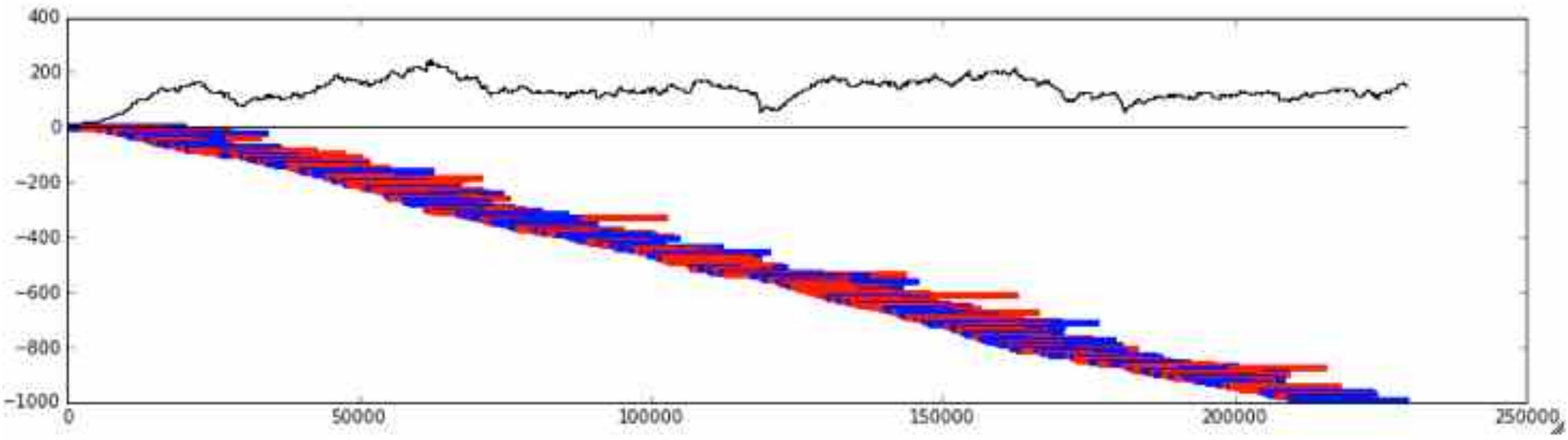
Plane Sweep Algorithm

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 %0.02f ms" % (brutetime)
print brutecov[0:10]
```

Add 1 to coverage vector at every position the read covers

```
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

Delta Encoding
aka run length encoding

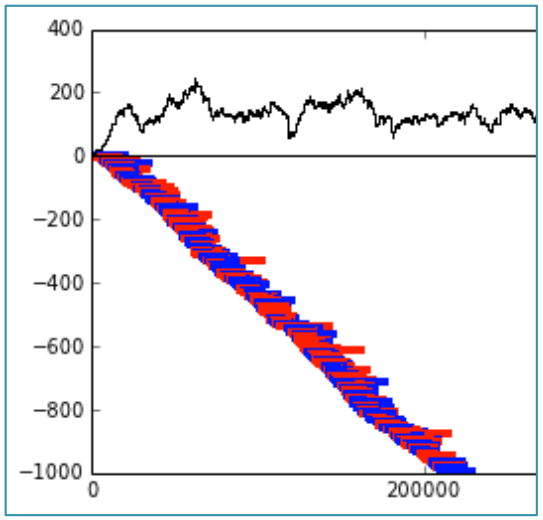
```
deltacov = []
curcov = -1
for i in xrange(0, len(brutecov)):
  if brutecov[i] != curcov:
    curcov = brutecov[i]
    delta = (i, curcov)
    deltacov.append(delta)

## 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 = deltacov[i][0]
  x2 = deltacov[i+1][0]
  y1  = YSCALE*deltacov[i][1]
  y2  = YSCALE*deltacov[i+1][1]

  ## draw the horizonal 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")
```

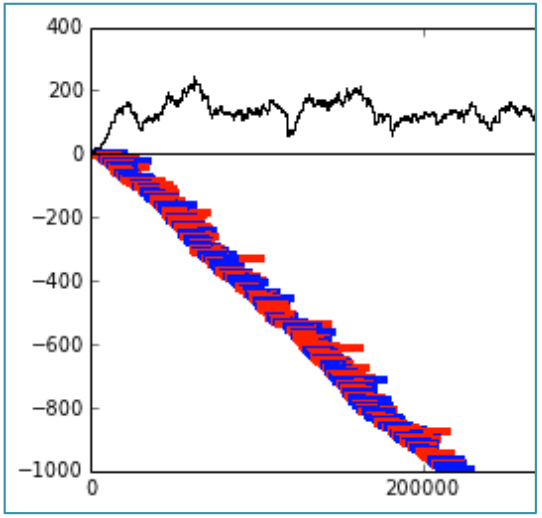


Plot Each Read

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 = deltacov[i][0]
  x2 = deltacov[i+1][0]
  y1  = YSCALE*deltacov[i][1]
  y2  = YSCALE*deltacov[i+1][

  ## draw the horizonal 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")
```



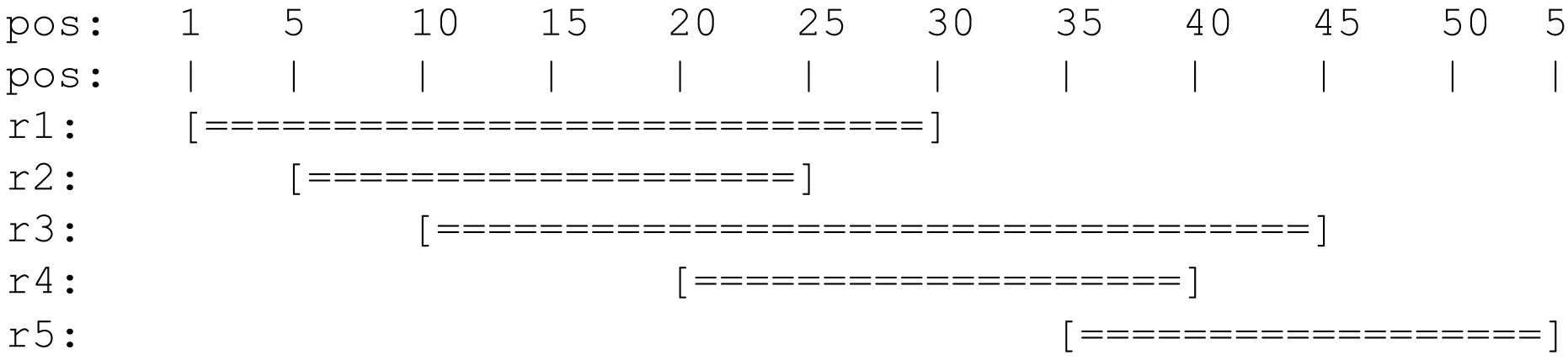
Plot Each Read

Brute Force works, but is pretty slow.

How can we make it go faster?

Plot Each Coverage Step

Plane Sweep



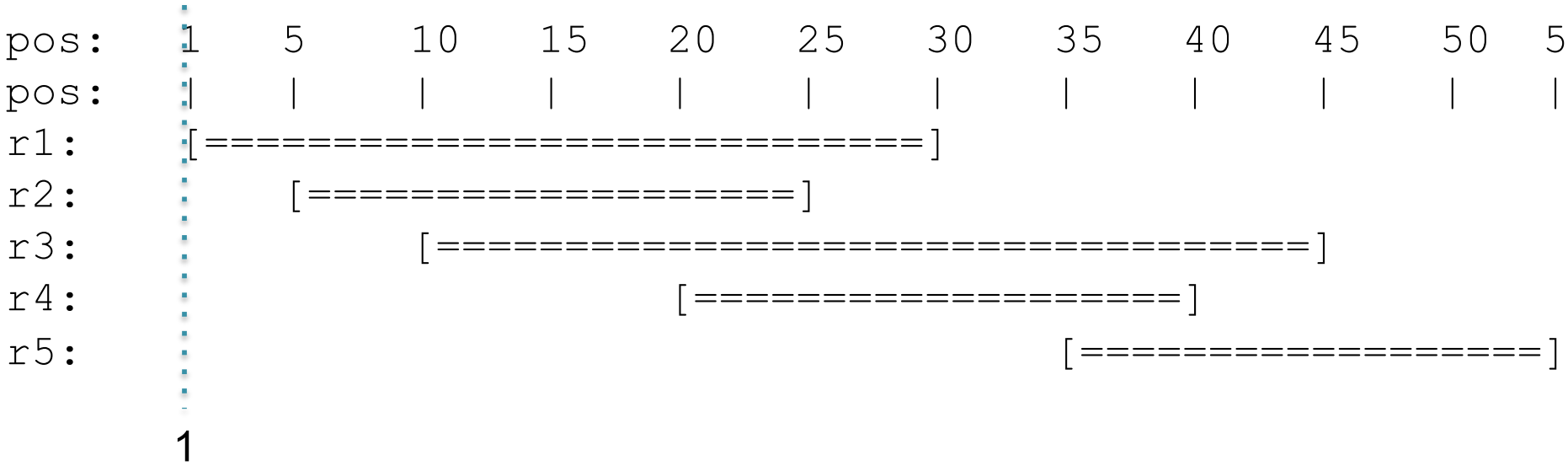
The basic algorithm works like this:

- 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

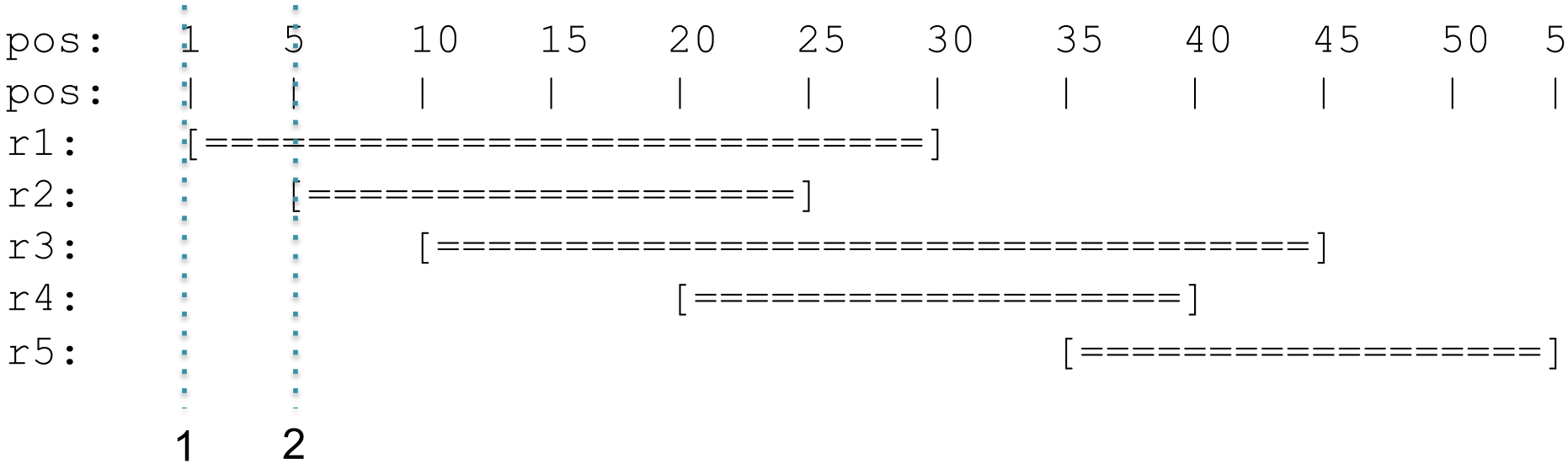


arrive at r1 [1,30]:

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

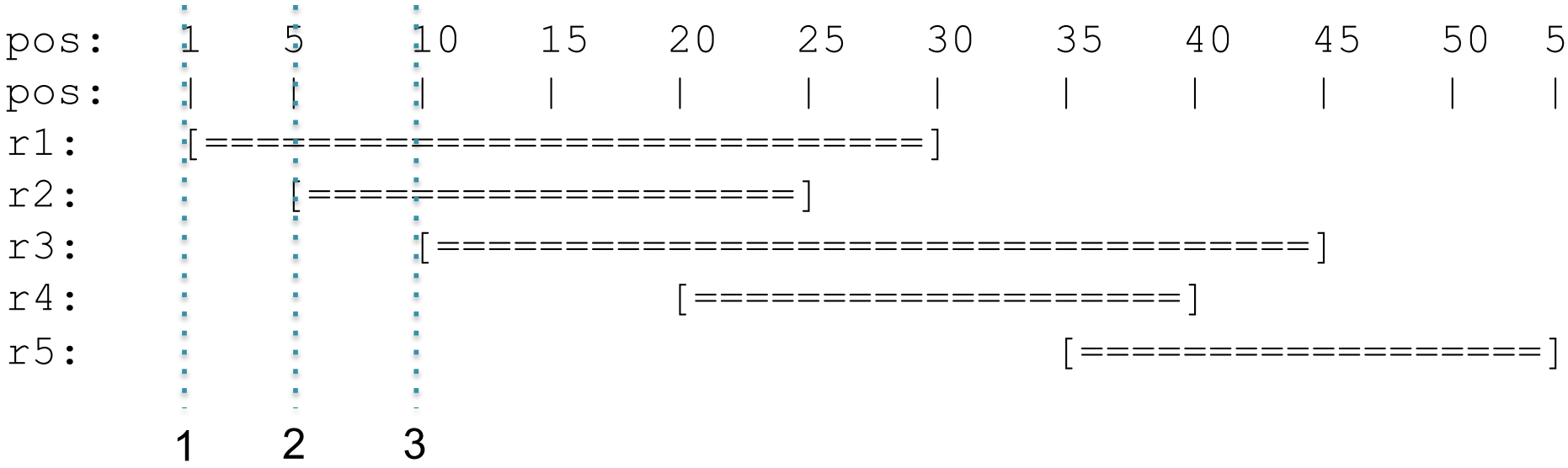
output (1,1)

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

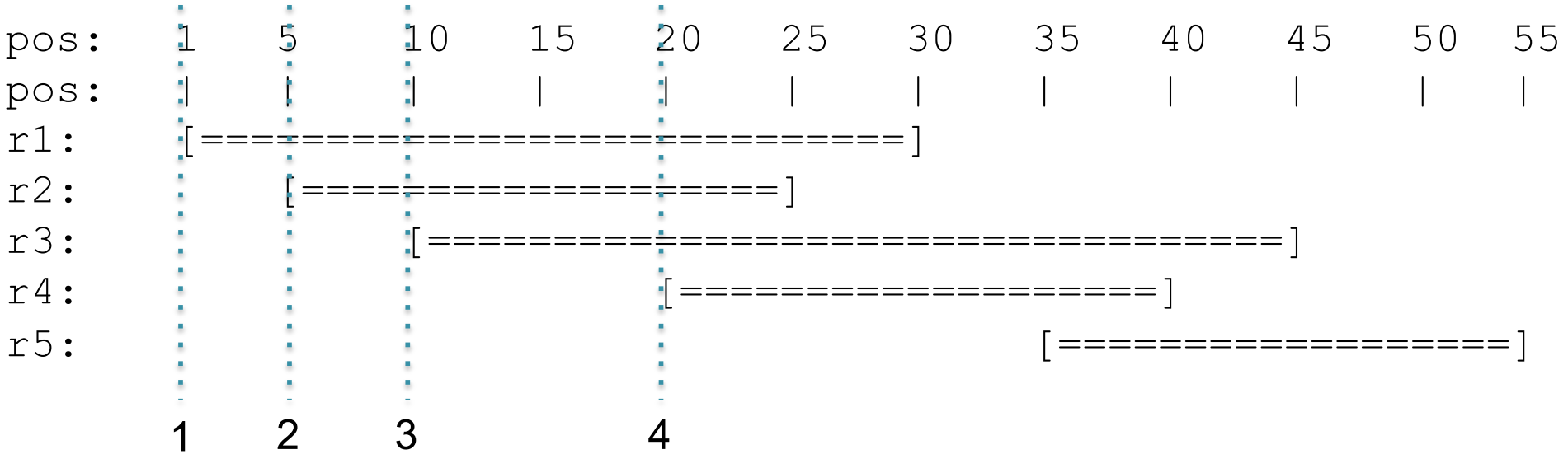
output (5, 2)

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

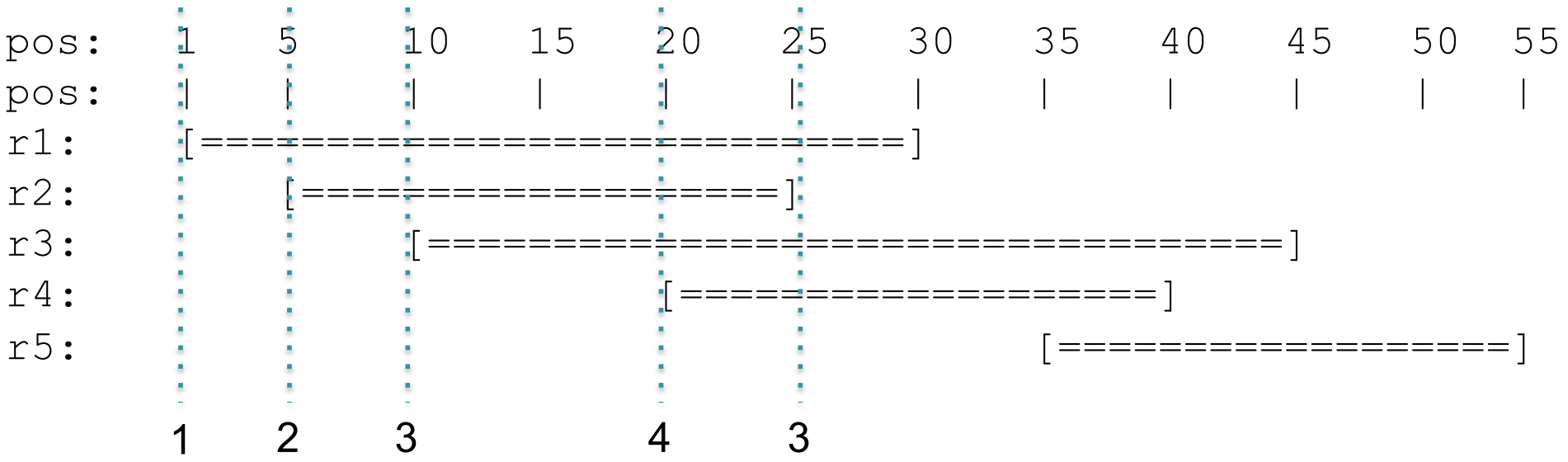
output (10, 3)

arrive at r4 [20,40]:

20 < 25; add to active set: 25, 30, 40, 45 <- out of order again

output (20, 4)

Plane Sweep

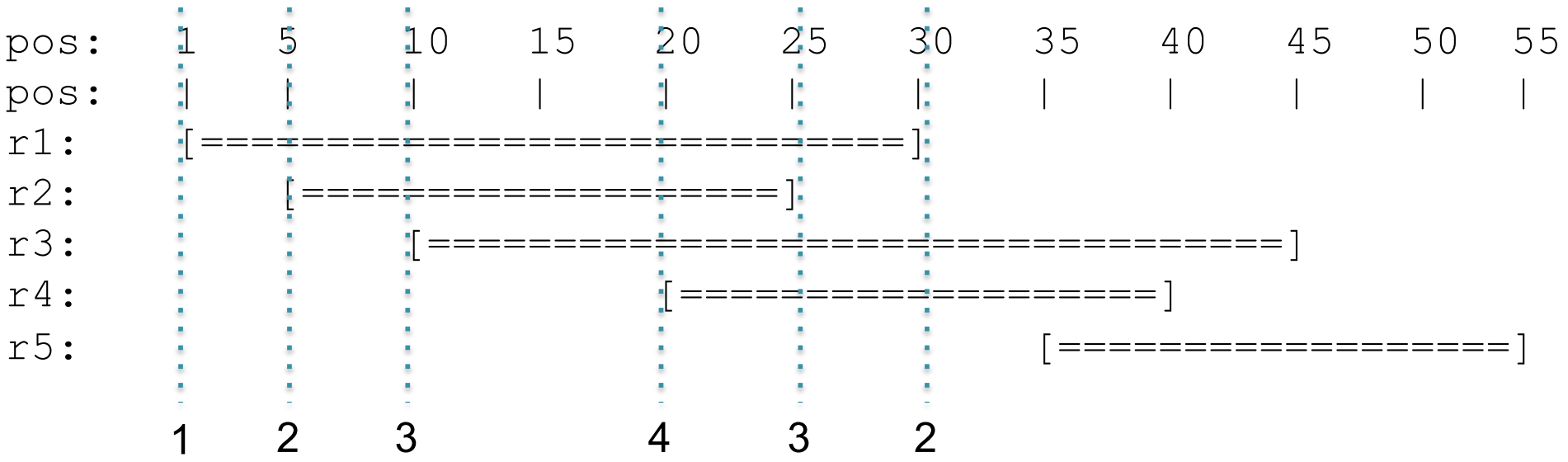


arrive at r5[35,55]:

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

output (25, 3)

Plane Sweep

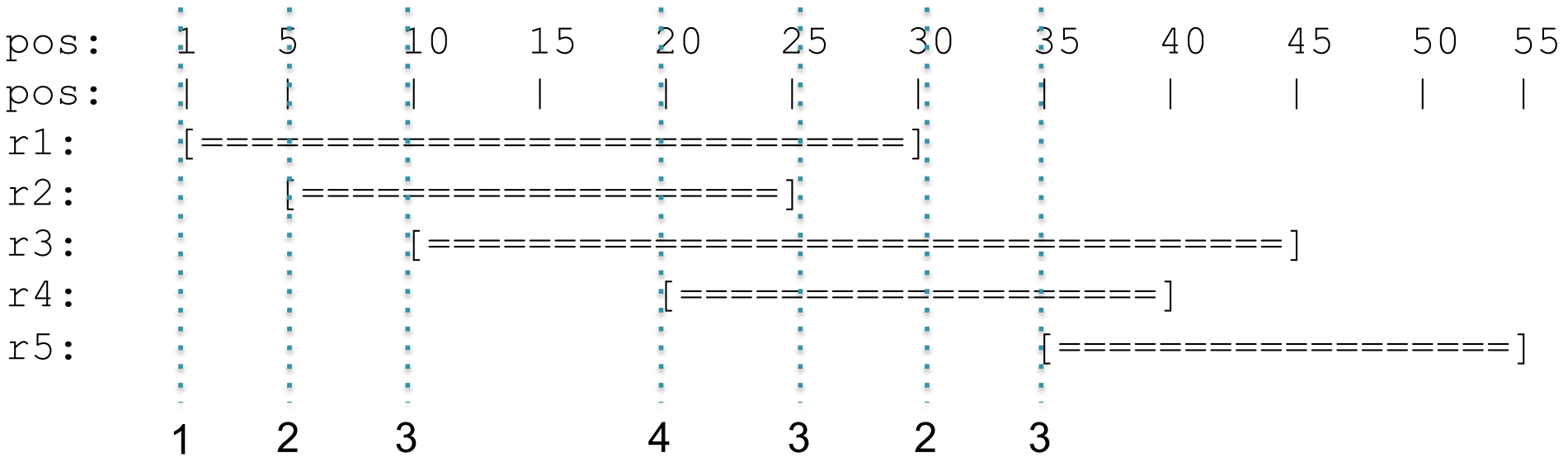


arrive at r5[35,55]:

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



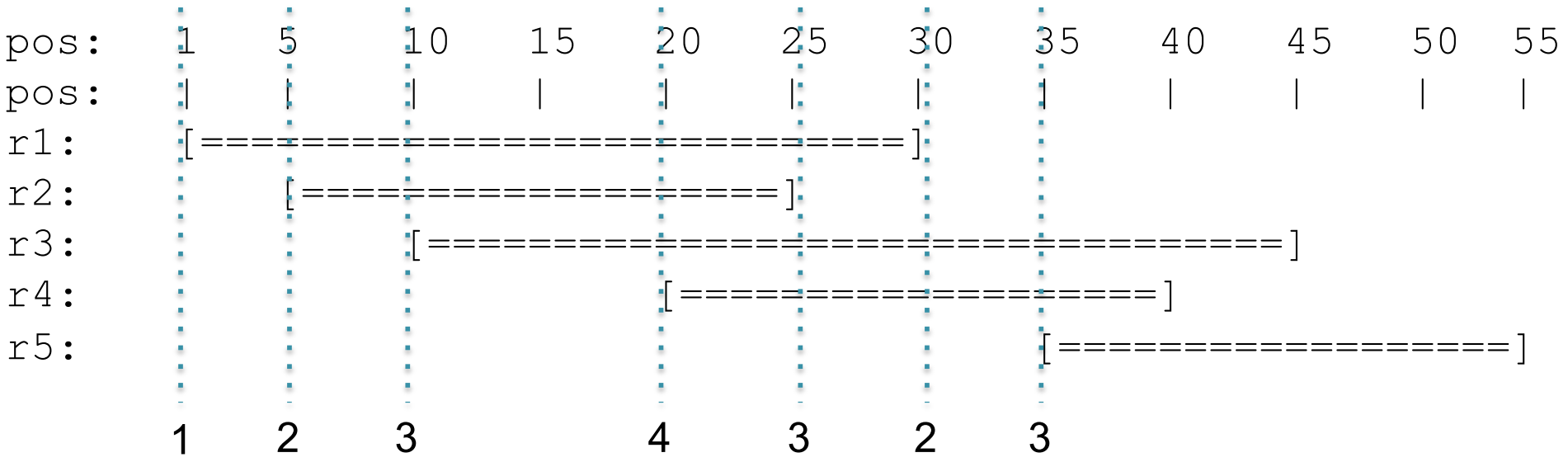
arrive at r5[35,55]:

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)

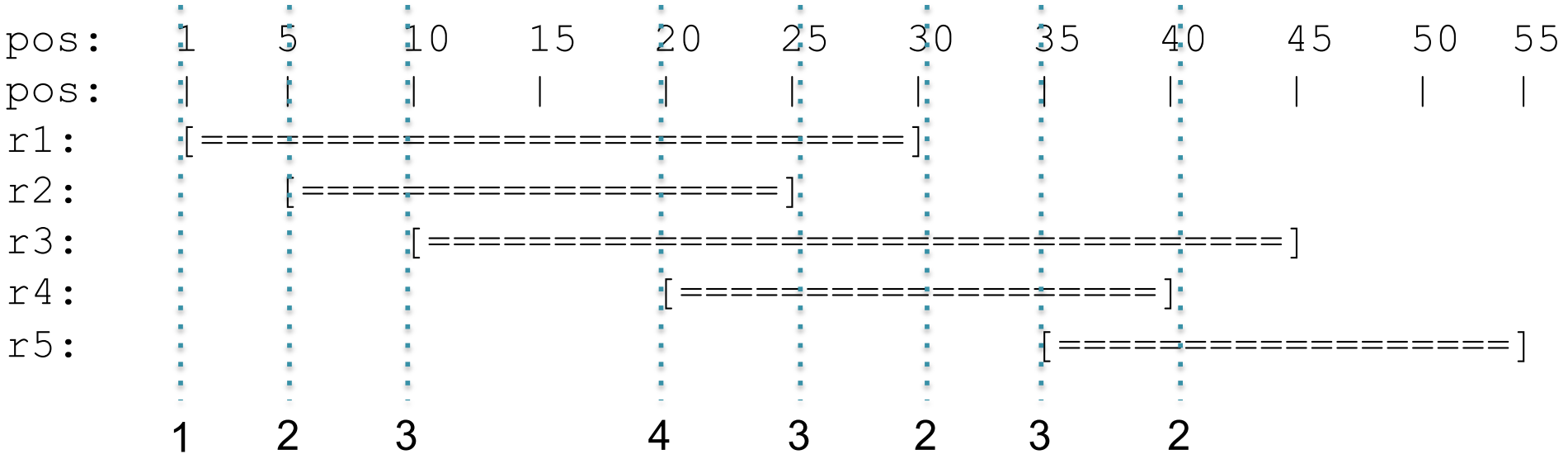
35 < 40: add to active set: 40, 45, 55
output (35, 3)

Plane Sweep



Flush:

Plane Sweep

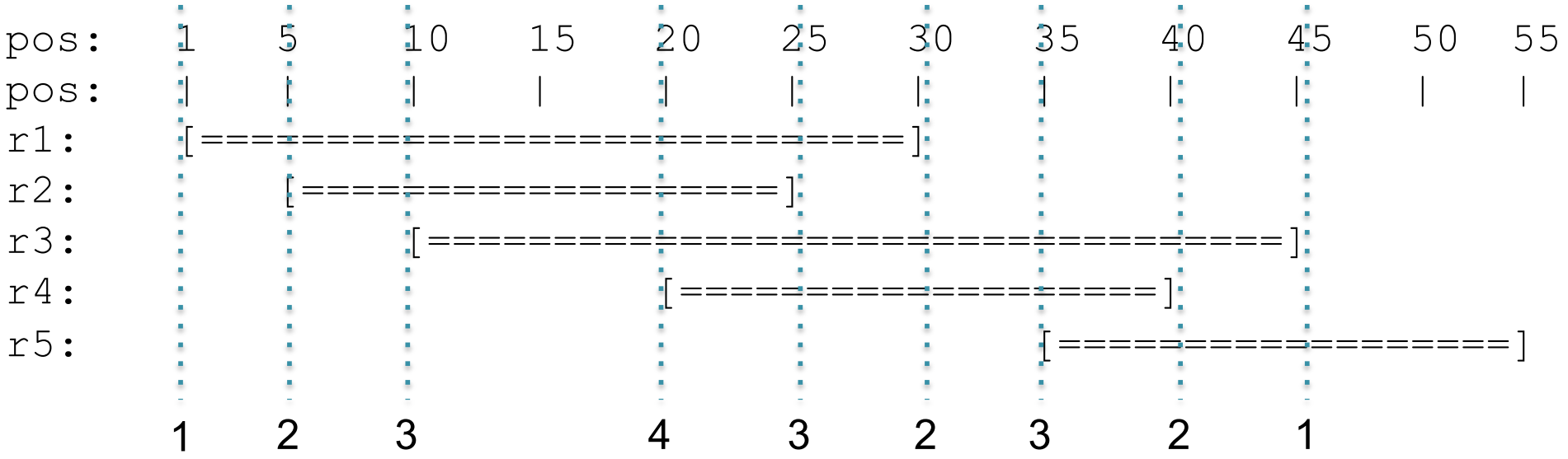


Flush:

step down at 40; active set: 45, 55

output (40, 2)

Plane Sweep

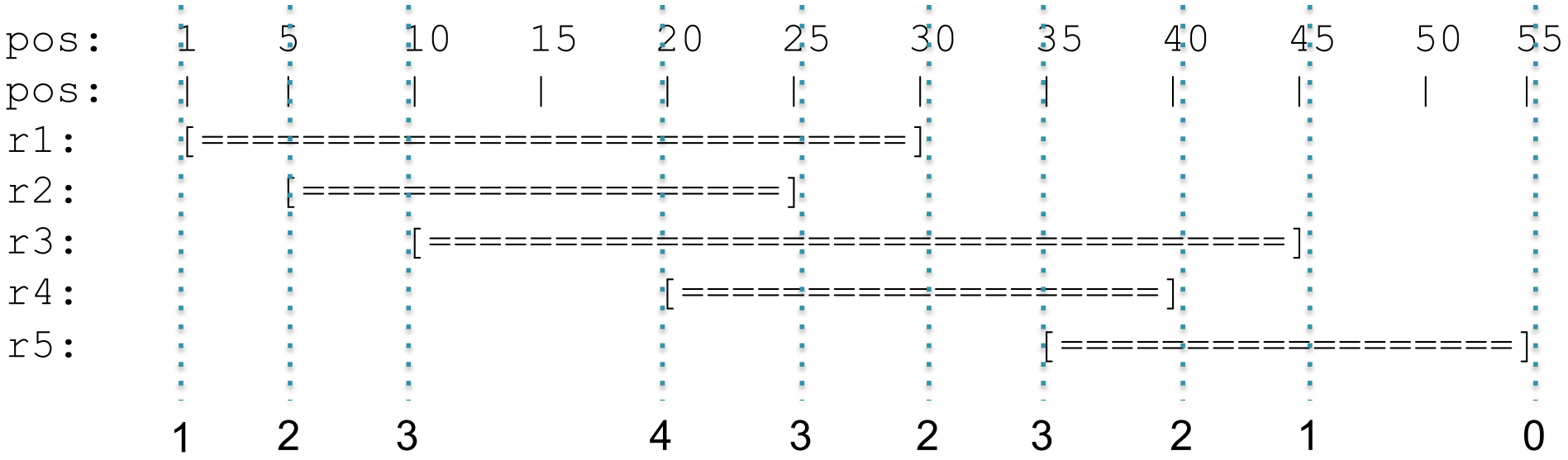


Flush:

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                                                    he list
  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)))
```

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

See notes on how to handle reads that have same coordinates

(Annoying bookkeeping :-/)

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

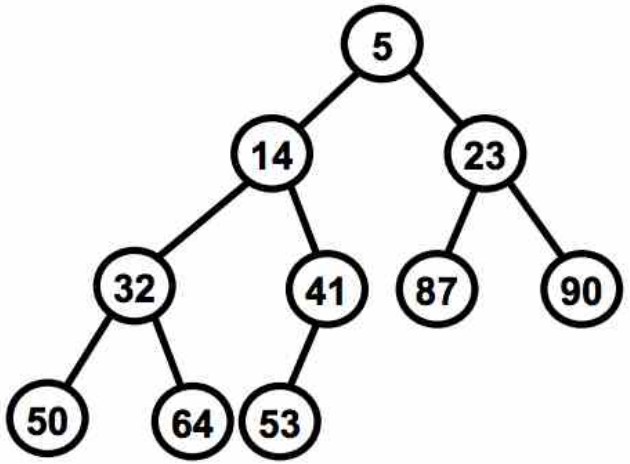
Do we really need the whole list to be sorted?

```
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

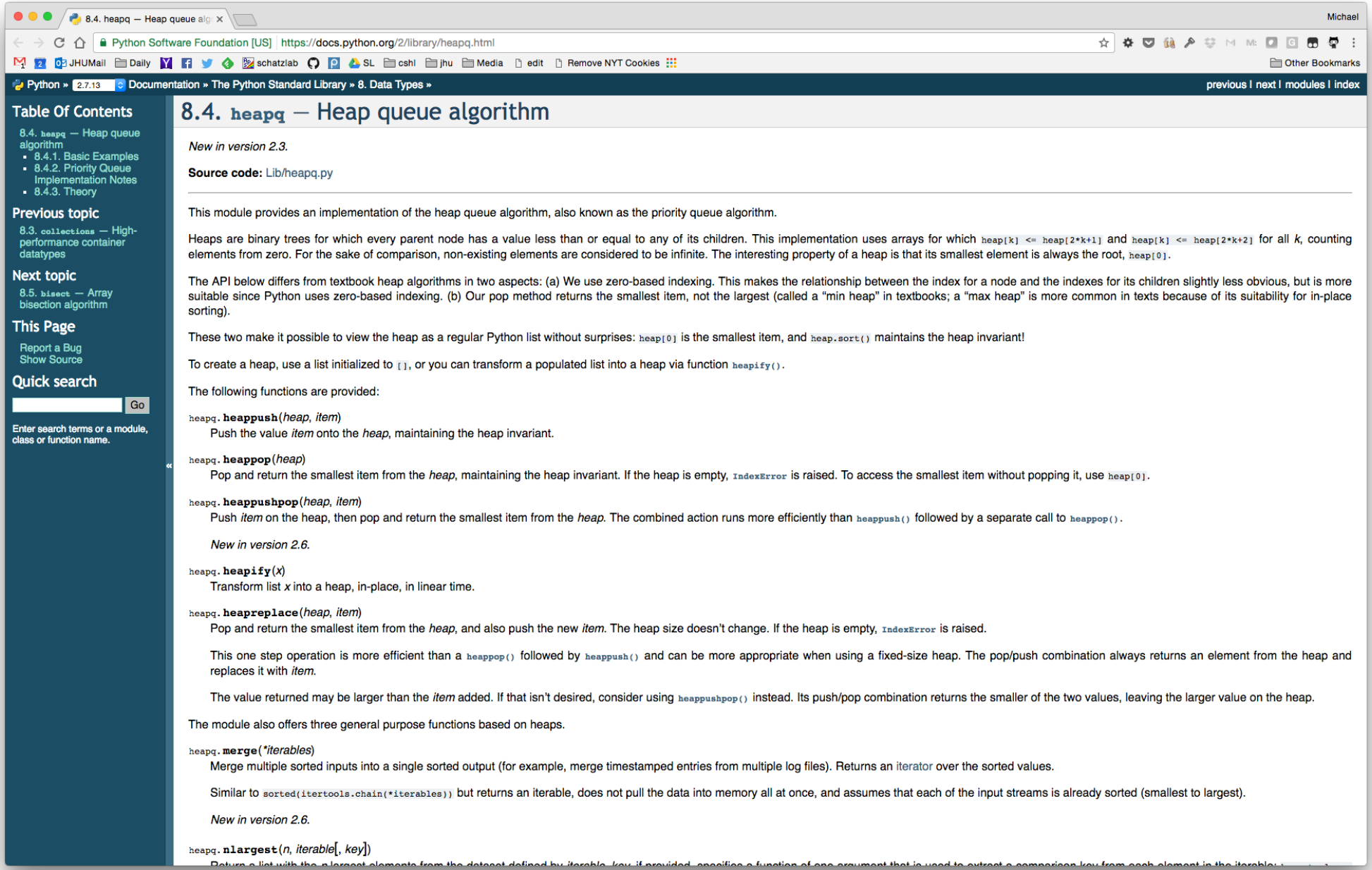


Very efficient data structure for dynamically maintaining a set of element while allowing you to find the minimum (or maximum) very fast:

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



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)):
        deltacovplane.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])):
    deltacovplane.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)
  deltacovplane.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)!
```