

Lecture 10. Genome Arithmetic

Michael Schatz

March 2, 2017

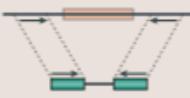
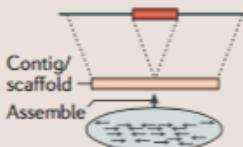
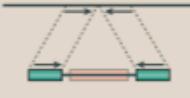
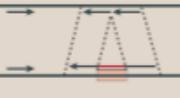
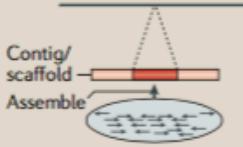
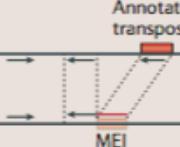
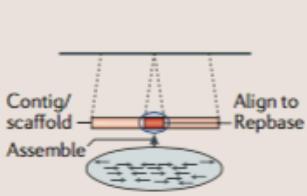
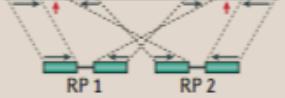
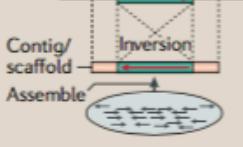
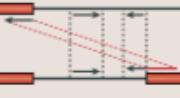
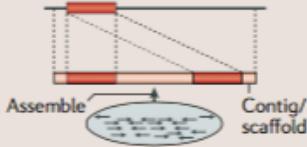
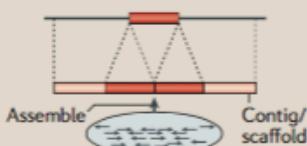
JHU 600.649: Applied Comparative Genomics



Part I: Structural Variation Review



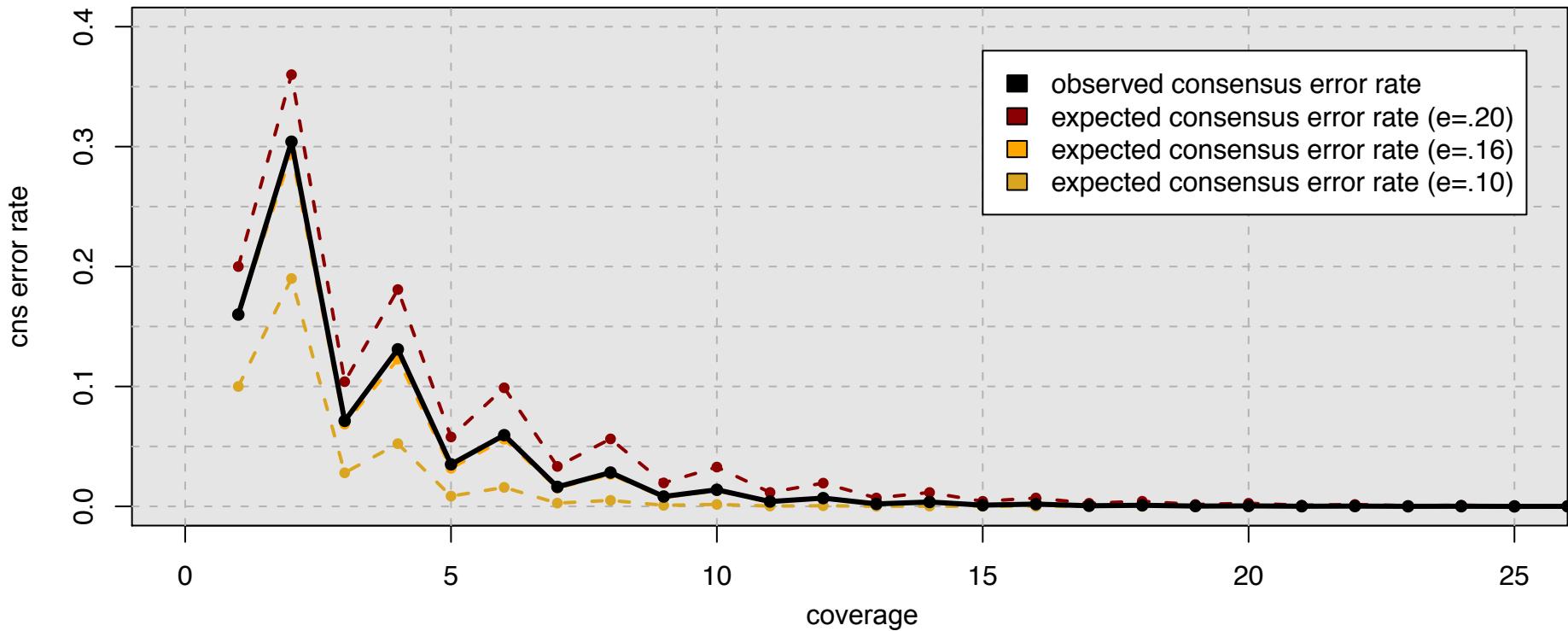
Structural Variation Sequence Signatures

SV classes	Read pair	Read depth	Split read	Assembly
Deletion				
Novel sequence insertion		Not applicable		
Mobile-element insertion		Not applicable		
Inversion		Not applicable		
Interspersed duplication				
Tandem duplication				

PacBio Single Molecule Real Time Sequencing (SMRT-sequencing)



Consensus Accuracy and Coverage

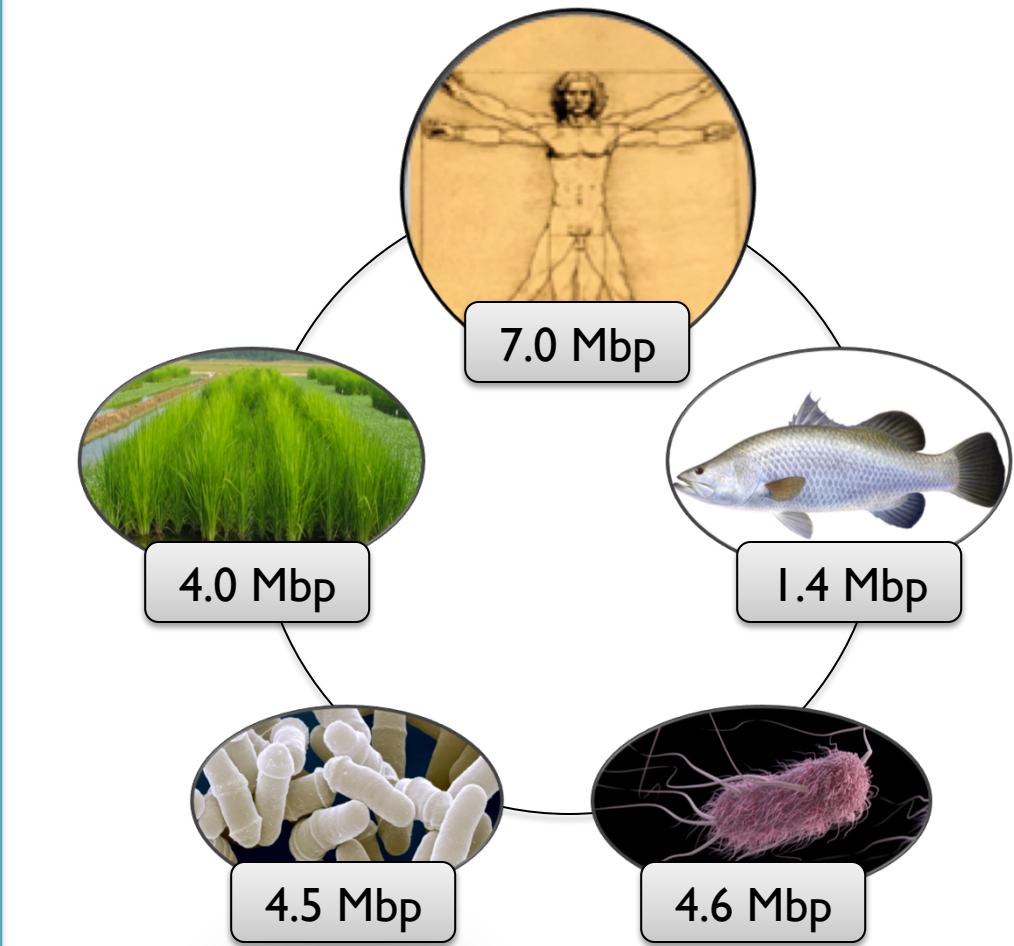
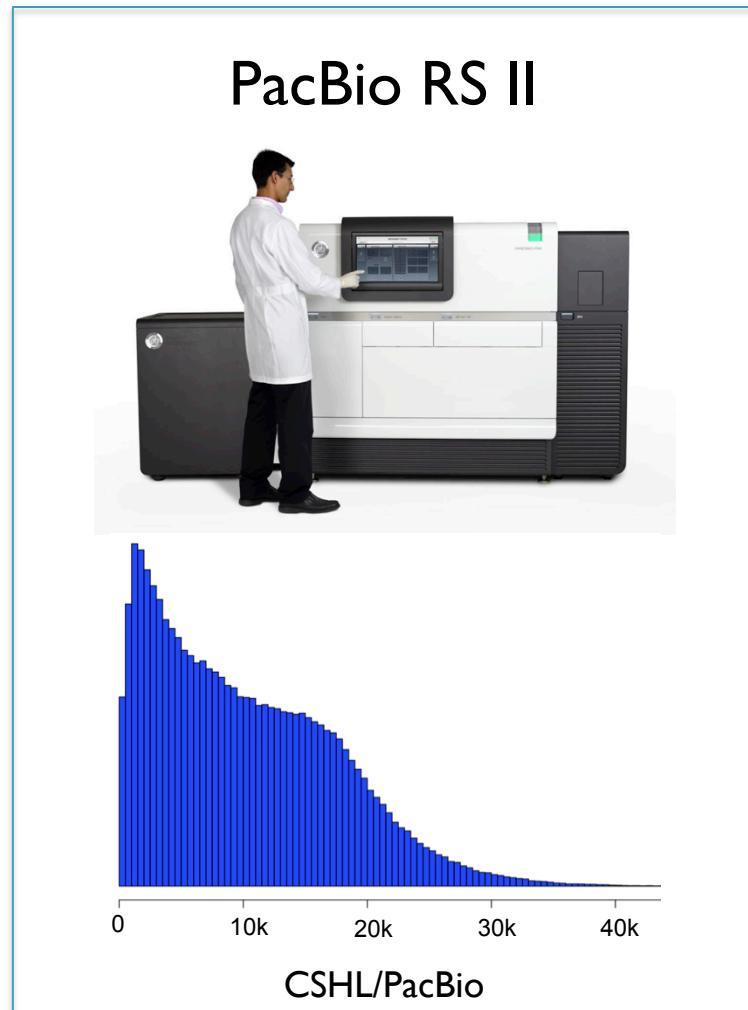


Coverage can overcome random errors

- Dashed: error model from binomial sampling; solid: observed accuracy
- For same reason, CCS is extremely accurate when using 5+ subreads

$$CNS\ Error = \sum_{i=\lceil c/2 \rceil}^c \binom{c}{i} (e)^i (1-e)^{n-i}$$

3rd Gen Long Read Sequencing



PacBio Roadmap



PacBio RS II

\$750k instrument cost
1895 lbs

~\$75k / human @ 50x



SMRTcell

150k Zero Mode Waveguides
~10kb average read length
~1 GB / SMRTcell
~\$500 / SMRTcell

PacBio Roadmap



PacBio Sequel

\$350k instrument cost
841 lbs

~\$30k / human @ 50x



SMRTcell v2

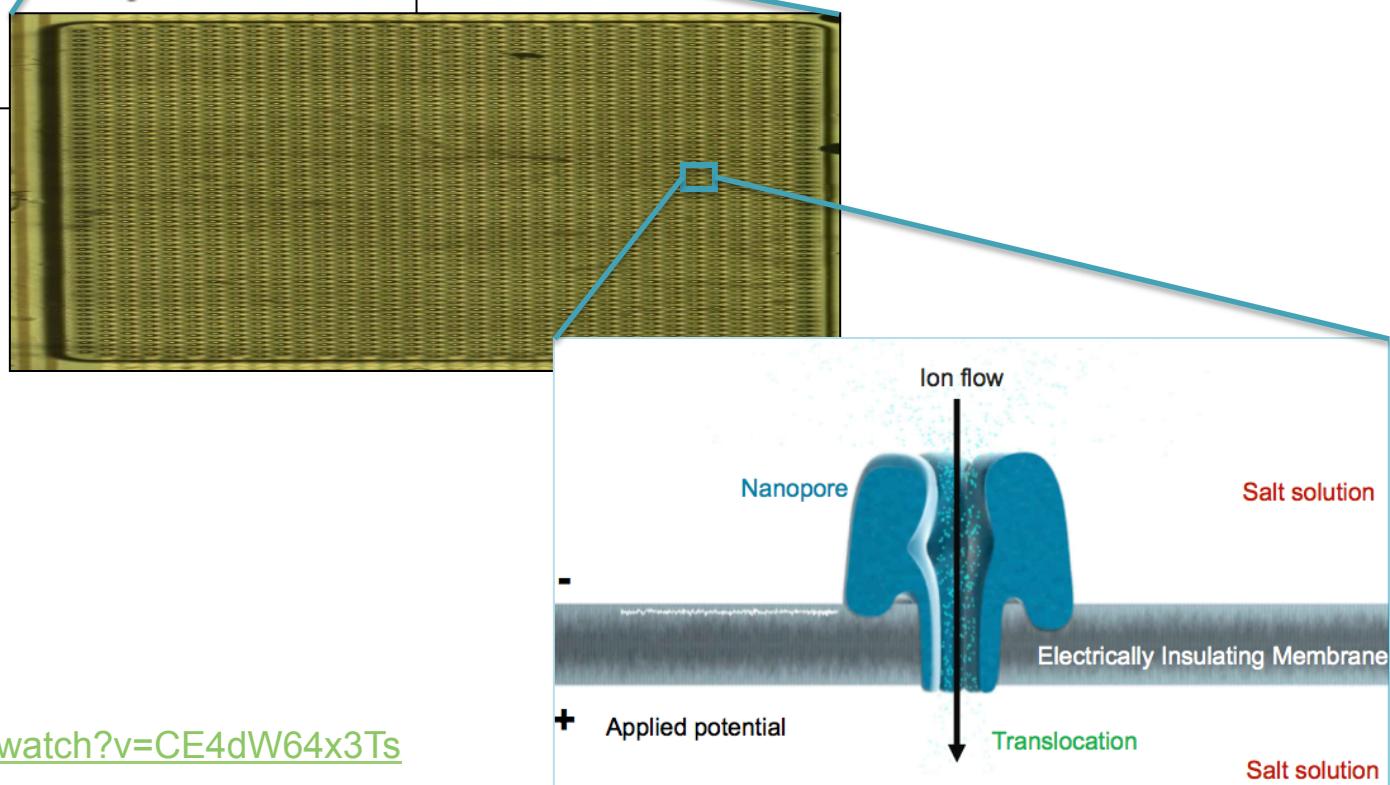
1M Zero Mode Waveguides
~15kb average read length
~10 GB / SMRTcell
~\$1000 / SMRTcell



Oxford Nanopore MinION



- Thumb drive sized sequencer powered over USB
- Capacity for 512 reads at once
- Senses DNA by measuring changes to ion flow



<https://www.youtube.com/watch?v=CE4dW64x3Ts>

Ebola Surveillance

LETTER

doi:10.1038/nature16996

Real-time, portable genome sequencing for Ebola surveillance

Joshua Quick^{1*}, Nicholas J. Loman^{1*}, Sophie Duraffour^{2,3*}, Jared T. Simpson^{4,5*}, Ettore Severi^{6*}, Lauren Cowley^{7*}, Joseph Akoi Bore², Raymond Koundouno², Gytis Dudas⁸, Amy Mikhail⁷, Nobila Ouédraogo⁹, Babak Afrough^{2,10}, Amadou Bah^{2,11}, Jonathan H. J. Baum^{2,3}, Beate Becker-Ziaja^{2,3}, Jan Peter Boettcher^{2,12}, Mar Cabeza-Cabrerizo^{2,3}, Álvaro Camino-Sánchez², Lisa L. Carter^{2,13}, Julianne Doerrbecker^{2,3}, Theresa Enkirch^{2,14}, Isabel García-Dorival^{2,15}, Nicole Hetzelt^{2,12}, Julia Hinzmann^{2,12}, Tobias Holm^{2,3}, Liana Eleni Kafetzopoulou^{2,16}, Michel Koropogui^{2,17}, Abigail Kosgey^{2,18}, Eeva Kuisma^{2,10}, Christopher H. Logue^{2,10}, Antonio Mazzarelli^{2,19}, Sarah Meisel^{2,3}, Marc Mertens^{2,20}, Janine Michel^{2,12}, Didier Ngabo^{2,10}, Katja Nitzsche^{2,3}, Elisa Pallasch^{2,3}, Livia Victoria Patrono^{2,3}, Jasmine Portmann^{2,21}, Johanna Gabriella Repits^{2,22}, Natasha Y. Rickett^{2,15,23}, Andreas Sachse^{2,12}, Katrin Singethan^{2,24}, Inés Vitoriano^{2,10}, Rahel L. Yemanaberhan^{2,3}, Elsa G. Zekeng^{2,15,23}, Trina Racine²⁵, Alexander Bello²⁵, Amadou Alpha Sall²⁶, Ousmane Faye²⁶, Oumar Faye²⁶, N'Faly Magassouba²⁷, Cecelia V. Williams^{28,29}, Victoria Amburgey^{28,29}, Linda Winona^{28,29}, Emily Davis^{29,30}, Jon Gerlach^{29,30}, Frank Washington^{29,30}, Vanessa Monteil³¹, Marine Jourdain³¹, Marion Bererd³¹, Alimou Camara³¹, Hermann Somlare³¹, Abdoulaye Camara³¹, Marianne Gerard³¹, Guillaume Bado³¹, Bernard Baillet³¹, Déborah Delaune^{32,33}, Koumpingnin Yacouba Nebie³⁴, Abdoulaye Diarra³⁴, Yacouba Savane³⁴, Raymond Bernard Pallawo³⁴, Giovanna Jaramillo Gutierrez³⁵, Natacha Milhano^{6,36}, Isabelle Roger³⁴, Christopher J. Williams^{6,37}, Facinet Yattara¹⁷, Kuiama Lewandowski¹⁰, James Taylor³⁸, Phillip Rachwal³⁸, Daniel J. Turner³⁹, Georgios Pollakis^{15,23}, Julian A. Hiscox^{15,23}, David A. Matthews⁴⁰, Matthew K. O'Shea⁴¹, Andrew McD. Johnston⁴¹, Duncan Wilson⁴¹, Emma Hutley⁴², Erasmus Smit⁴³, Antonino Di Caro^{2,19}, Roman Wölfe^{2,44}, Kilian Stoecker^{2,44}, Erna Fleischmann^{2,44}, Martin Gabriel^{2,3}, Simon A. Weller³⁸, Lamine Koivogui¹⁴⁵, Boubacar Diallo³⁴, Sakoba Keïta¹⁷, Andrew Rambaut^{8,46,47}, Pierre Formenty³⁴, Stephan Günther^{2,3} & Miles W. Carroll^{2,10,48,49}

Ebola Surveillance

LETTER

doi:10.1038/nature16996

Real-time, portable genome sequencing for Ebola surveillance

Joshua Quick^{1*}, Nicholas J. Loman^{1*}, Sophie Duraffour^{2,3*}, Jared T. Simpson^{4,5*}, Ettore Se Joseph Akoi Bore², Raymond Koundouno², Gytis Dudas⁸, Amy Mikhail⁷, Nobila Ouédraogo Amadou Bah^{2,11}, Jonathan H. J. Baum^{2,3}, Beate Becker-Ziaja^{2,3}, Jan Peter Boettcher^{2,12}, Mar Álvaro Camino-Sánchez², Lisa L. Carter^{2,13}, Julianne Doerrbecker^{2,3}, Theresa Enkirch^{2,14}, Is Nicole Hetzelt^{2,12}, Julia Hinzmann^{2,12}, Tobias Holm^{2,3}, Liana Eleni Kafetzopoulou^{2,16}, Michelle Eeva Kuisma^{2,10}, Christopher H. Logue^{2,10}, Antonio Mazzarelli^{2,19}, Sarah Meisel^{2,3}, Marc M Didier Ngabo^{2,10}, Katja Nitzsche^{2,3}, Elisa Pallasch^{2,3}, Livia Victoria Patrono^{2,3}, Jasmine Port Natasha Y. Rickett^{2,15,23}, Andreas Sachse^{2,12}, Katrin Singethan^{2,24}, Inés Vitoriano^{2,10}, Rahel Elsa G. Zekeng^{2,15,23}, Trina Racine²⁵, Alexander Bello²⁵, Amadou Alpha Sall²⁶, Ousmane Fa N'Faly Magassouba²⁷, Cecelia V. Williams^{28,29}, Victoria Amburgey^{28,29}, Linda Winona^{28,29}, Eric Frank Washington^{29,30}, Vanessa Monteil³¹, Marine Jourdain³¹, Marion Bererd³¹, Alimou Cam Abdoulaye Camara³¹, Marianne Gerard³¹, Guillaume Bado³¹, Bernard Baillet³¹, Déborah Dela Abdoulaye Diarra³⁴, Yacouba Savane³⁴, Raymond Bernard Pallawo³⁴, Giovanna Jaramillo Gu Isabelle Roger³⁴, Christopher J. Williams^{6,37}, Facinet Yattara¹⁷, Kuiama Lewandowski¹⁰, James Daniel J. Turner³⁹, Georgios Pollakis^{15,23}, Julian A. Hiscox^{15,23}, David A. Matthews⁴⁰, Matthew Andrew McD. Johnston⁴¹, Duncan Wilson⁴¹, Emma Hutley⁴², Erasmus Smit⁴³, Antonino Di Giacomo⁴⁴, Kilian Stoecker^{2,44}, Erna Fleischmann^{2,44}, Martin Gabriel^{2,3}, Simon A. Weller³⁸, Lamine Koi⁴⁵, Sakoba Keïta¹⁷, Andrew Rambaut^{8,46,47}, Pierre Formenty³⁴, Stephan Günther^{2,3} & Miles W. O'Connor¹



Figure 1 | Deployment of the portable genome surveillance system in Guinea. **a**, We were able to pack all instruments, reagents and disposable consumables within aircraft baggage. **b**, We initially established the genomic surveillance laboratory in Donka Hospital, Conakry, Guinea. **c**, Later we moved the laboratory to a dedicated sequencing laboratory in Coyah prefecture. **d**, Within this laboratory we separated the sequencing instruments (on the left) from the PCR bench (to the right). An uninterruptable power supply can be seen in the middle that provides power to the thermocycler. (Photographs taken by J.Q. and S.D.)

Ebola Surveillance

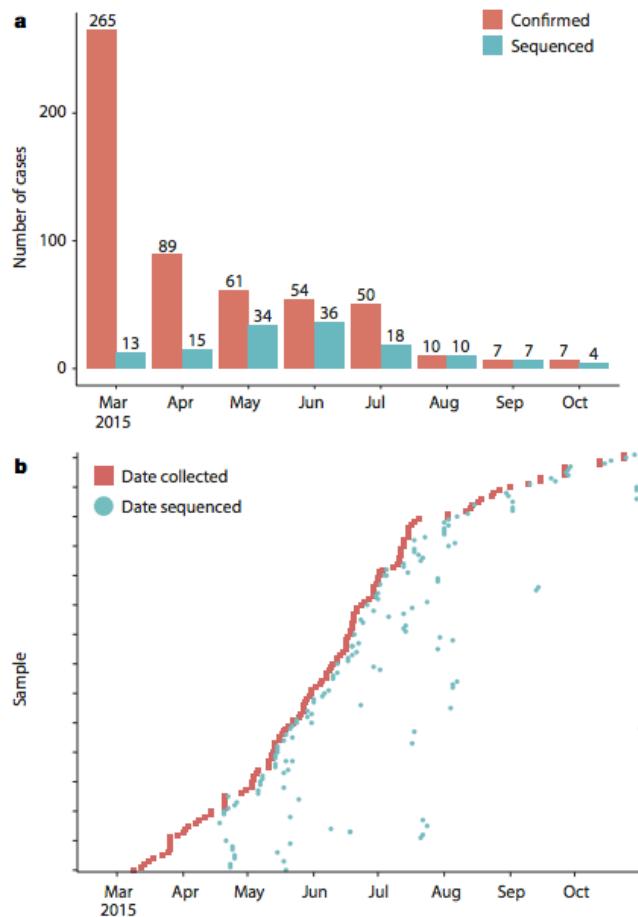


Figure 2 | Real-time genomics surveillance in context of the Guinea Ebola virus disease epidemic. a, Here we show the number of reported cases of Ebola virus disease in Guinea (red) in relation to the number of EBOV new patient samples ($n = 137$, in blue) generated during this study. b, For each of the 142 sequenced samples, we show the relationship between sample collection date (red) and the date of sequencing (blue). Twenty-eight samples were sequenced within three days of the sample being taken, and sixty-eight samples within a week. Larger gaps represent retrospective sequencing of cases to provide additional epidemiological context.

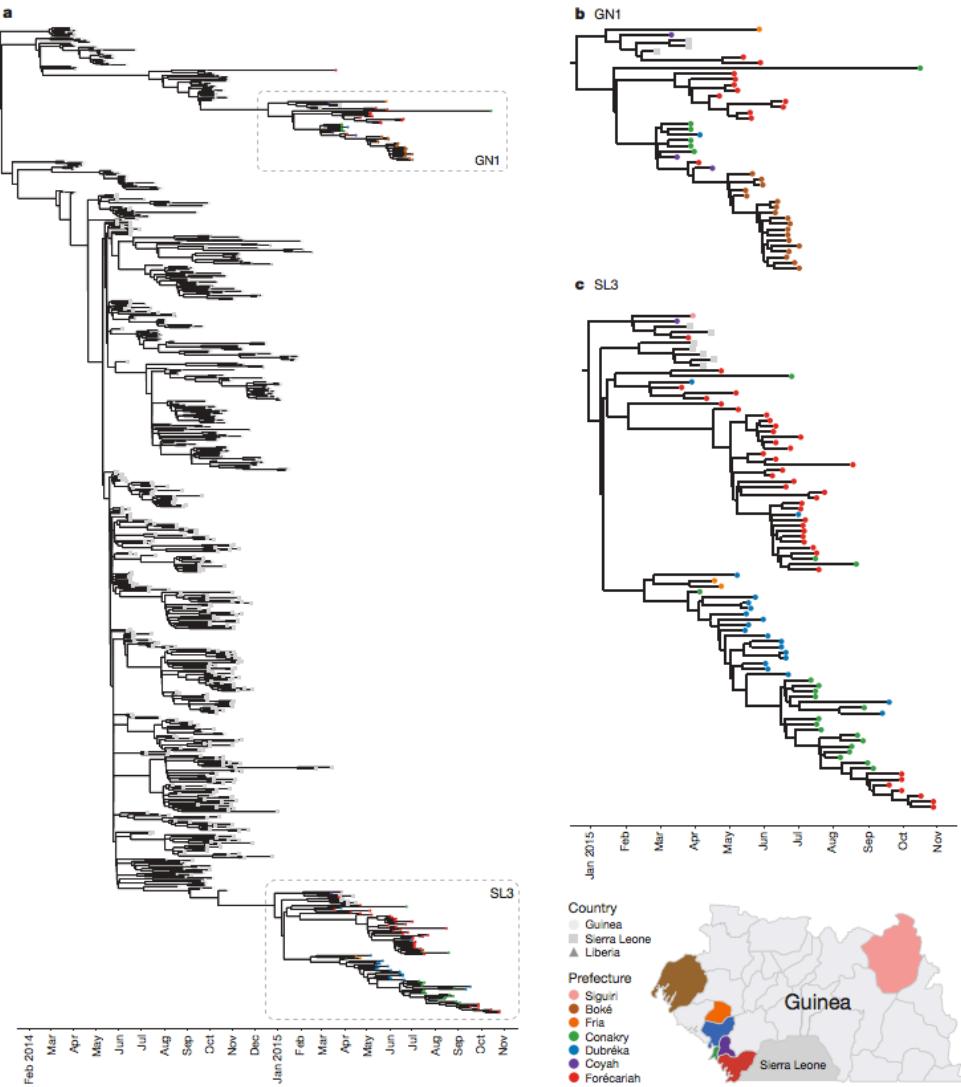
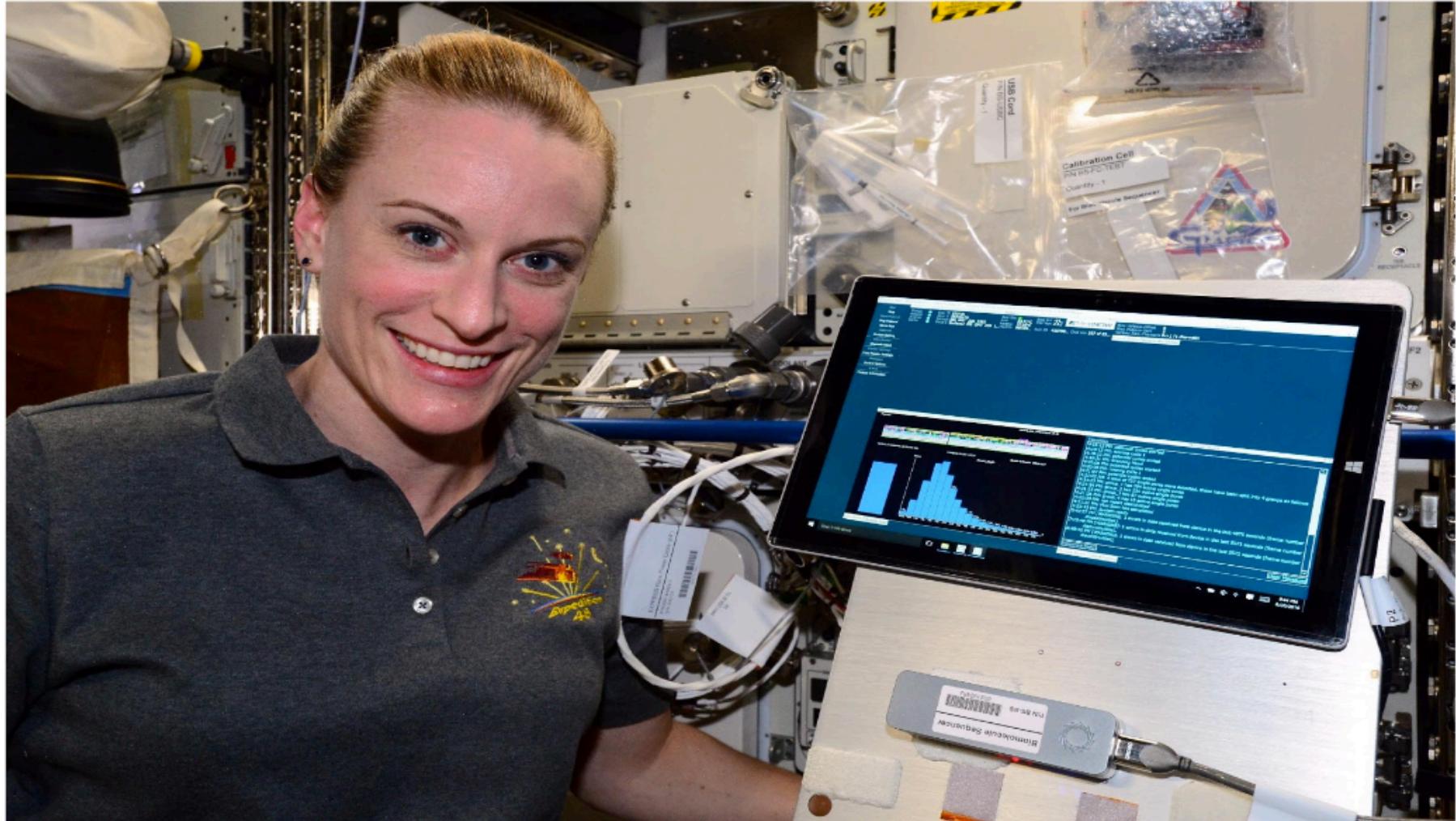


Figure 3 | Evolution of EBOV over the course of the Ebola virus disease epidemic. a, Time-scaled phylogeny of 603 published sequences with 125 high quality sequences from this study. The shape of nodes on the tree demonstrates country of origin. Our results show Guinean samples (coloured circles) belong to two previously identified lineages, GN1 and SL3. b, GN1 is deeply branching with early epidemic samples. c, SL3 is

related to cases identified in Sierra Leone. Samples are frequently clustered by geography (indicated by colour of circle) and this provides information as to origins of new introductions, such as in the Boké epidemic in May 2015. Map figure adapted from SimpleMaps website (<http://simplemaps.com/resources/svg-gn>).



Extremely Portable Sequencing!



Kate Rubins sequencing DNA on the ISS

Oxford Nanopore



MinION

\$2k / instrument
1-2 GB / day
~\$300k / human @ 50x

PromethION

\$75k / instrument
>>100GB / day
??? / human @ 50x

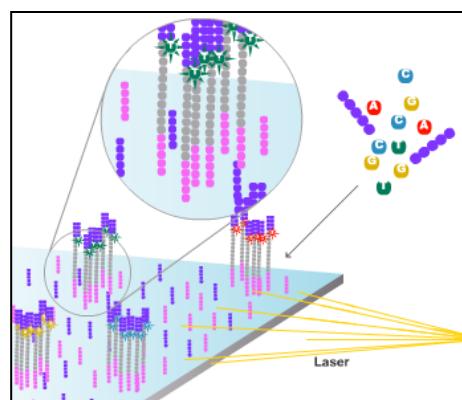
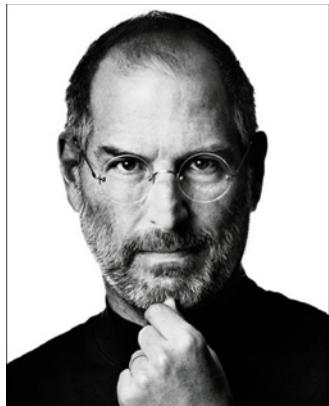
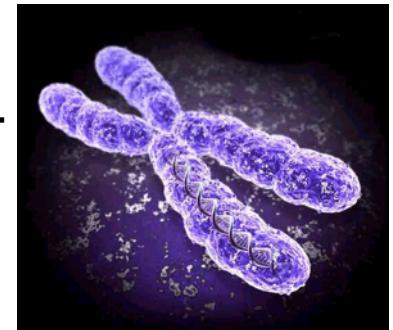
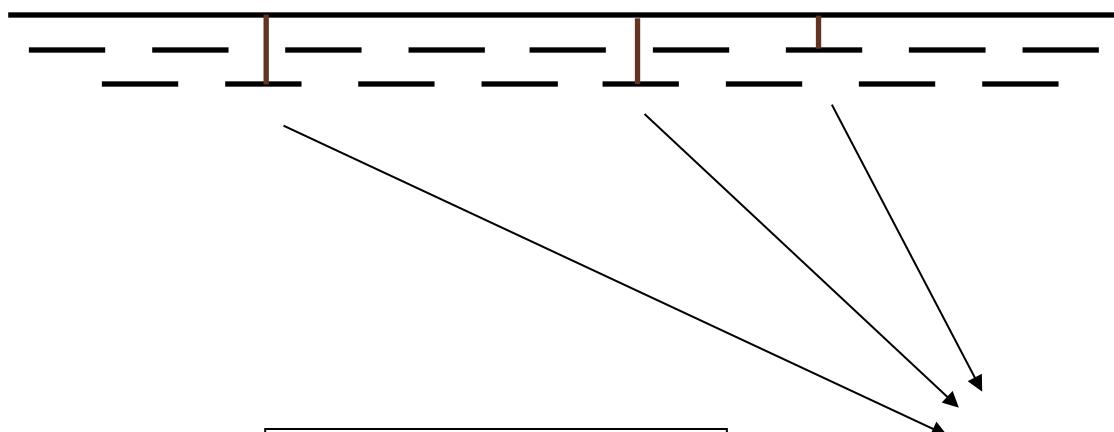
Oxford Nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome
Goodwin, S, Gurtowski, J, Ethe-Sayers, S, Deshpande, P, Schatz MC* McCombie, WR* (2015) Genome Research doi: 10.1101/gr.191395.115

Part 2: Genome Arithmetic



Personal Genomics

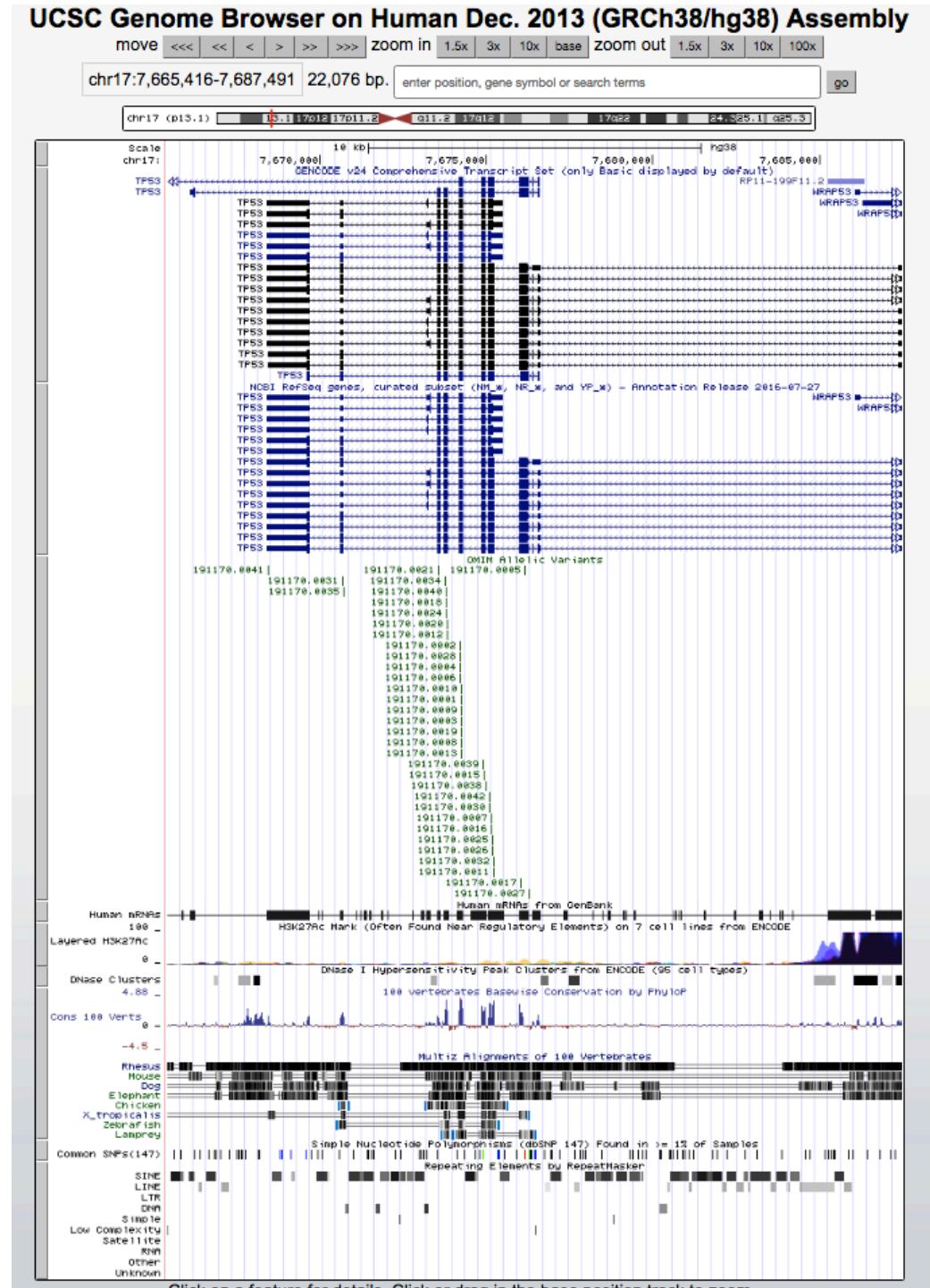
How does your genome compare to the reference?



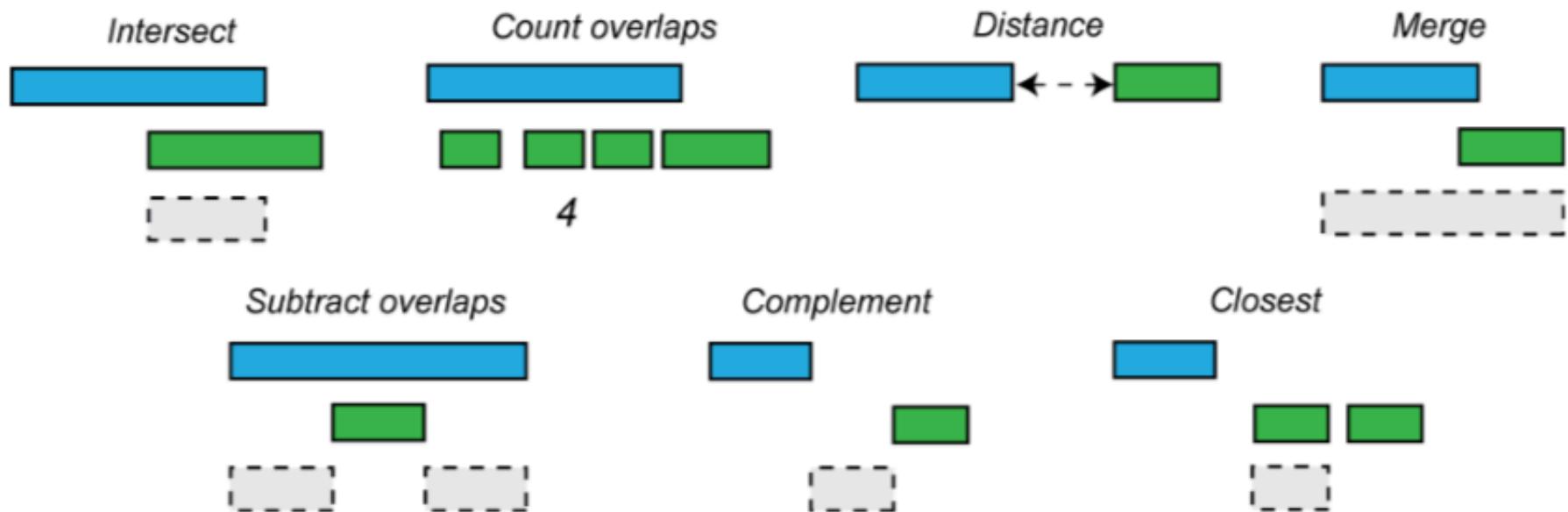
Heart Disease
Cancer
Creates magical technology

What are genome intervals?

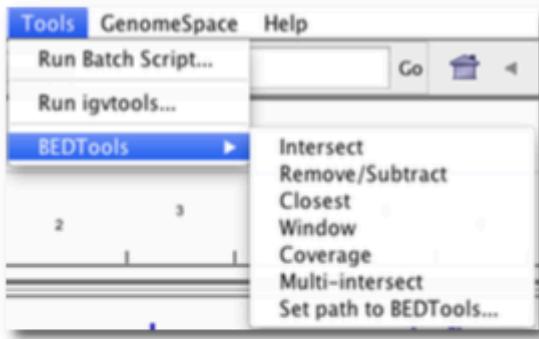
- Genetic variation:
 - SNPs: 1bp
 - Indels: 1-50bp
 - SVs: >50bp
- Genes:
 - exons, introns, UTRs, promoters
- Conservation
- Transposons
- Origins of replication
- TF binding sites
- CpG islands
- Segmental duplications
- Sequence alignments
- Chromatin annotations
- Gene expression data
- ...
- ***Your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



BEDTools to the rescue!



Getting & Using BEDTools



Integrated into IGV

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

A screenshot of the bedtools documentation page from readthedocs. The title is 'bedtools v2.26.0'. The main content features a large red logo with a white shield containing a stylized 'b'. Below the logo, the text reads: 'bedtools is a fast, flexible toolset for genome arithmetic.' It describes bedtools as a 'swiss-army knife of tools for a wide-range of genomics analysis tasks'. The page includes sections for 'Bedtools links', 'Sources', 'Releases', and 'This Page'. A 'Quick search' bar is at the bottom left, and a 'Table of contents' link is at the bottom right. The URL in the browser is 'bedtools.readthedocs.io/en/latest/'.

Extensive Documentation and Examples

Genomic Coordinates

What are coordinates of “TAC”
in GATTACA?

1-based coordinates

- Base 4 through 6: [4,6] “closed”
- Base 4 through 7: [4,7) “half-open”
- 3 bases starting at base 4: [4, +3]

GAT**TAC**A
1234567

0-based coordinates

- Position 3 through 5: [3,5] “closed”
- Position 3 through 6: [3,6) “half-open”
- 3 bases starting at position 3: [3, +3]

GAT**TAC**A
0123456

Genomic Conventions

1-based coordinates

- BLAST/MUMmer alignments
- Ensembl Genome Browser
- SAM,VCF, GFF and Wiggle

GAT^TAC
1234567

0-based coordinates

- BAM, BCFv2, BED, and PSL
- UCSC Genome Browser
- C/C++, Perl, Python, Java

GAT^TAC
0123456

Always double check the manual!
You will get this wrong someday 😞

BED Format

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

The first three required BED fields are:

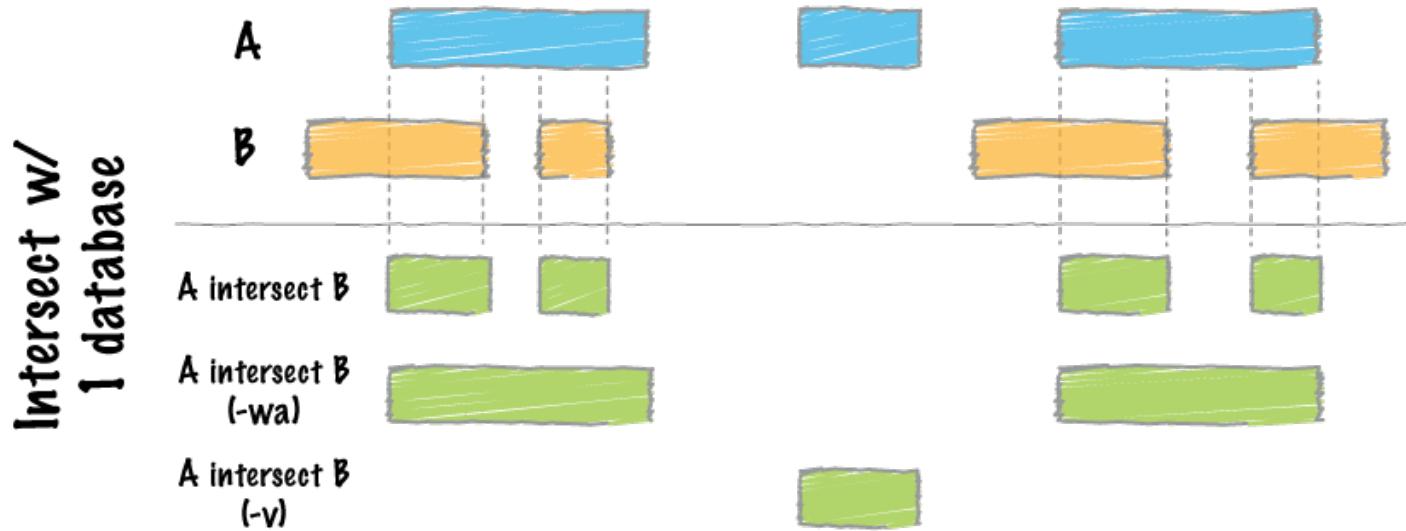
1. chrom The name of the chromosome (e.g. chr3, chrY, chr2_random) or scaffold (e.g. scaffold10671).
2. chromStart The starting position of the feature in the chromosome or scaffold. The first base in a sequence is numbered 0.
3. chromEnd The ending position of the feature in the chromosome or scaffold.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



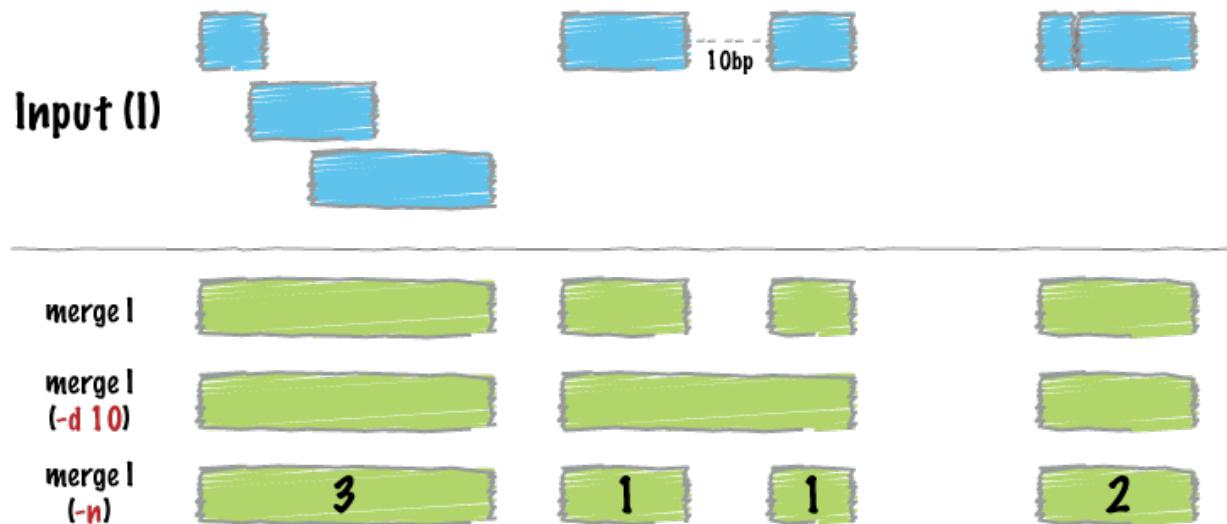
What exons are hit by SVs?

```
$ 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



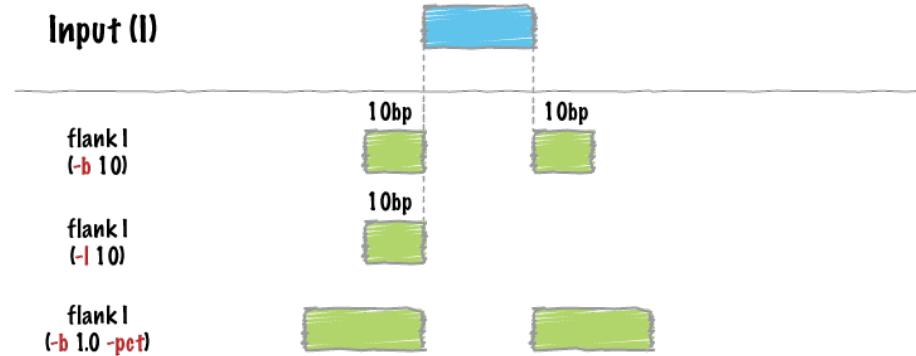
What parts of the genome are exonic?

```
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   17268
```

Note input must be sorted!

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

BEDTools Flank & getfasta



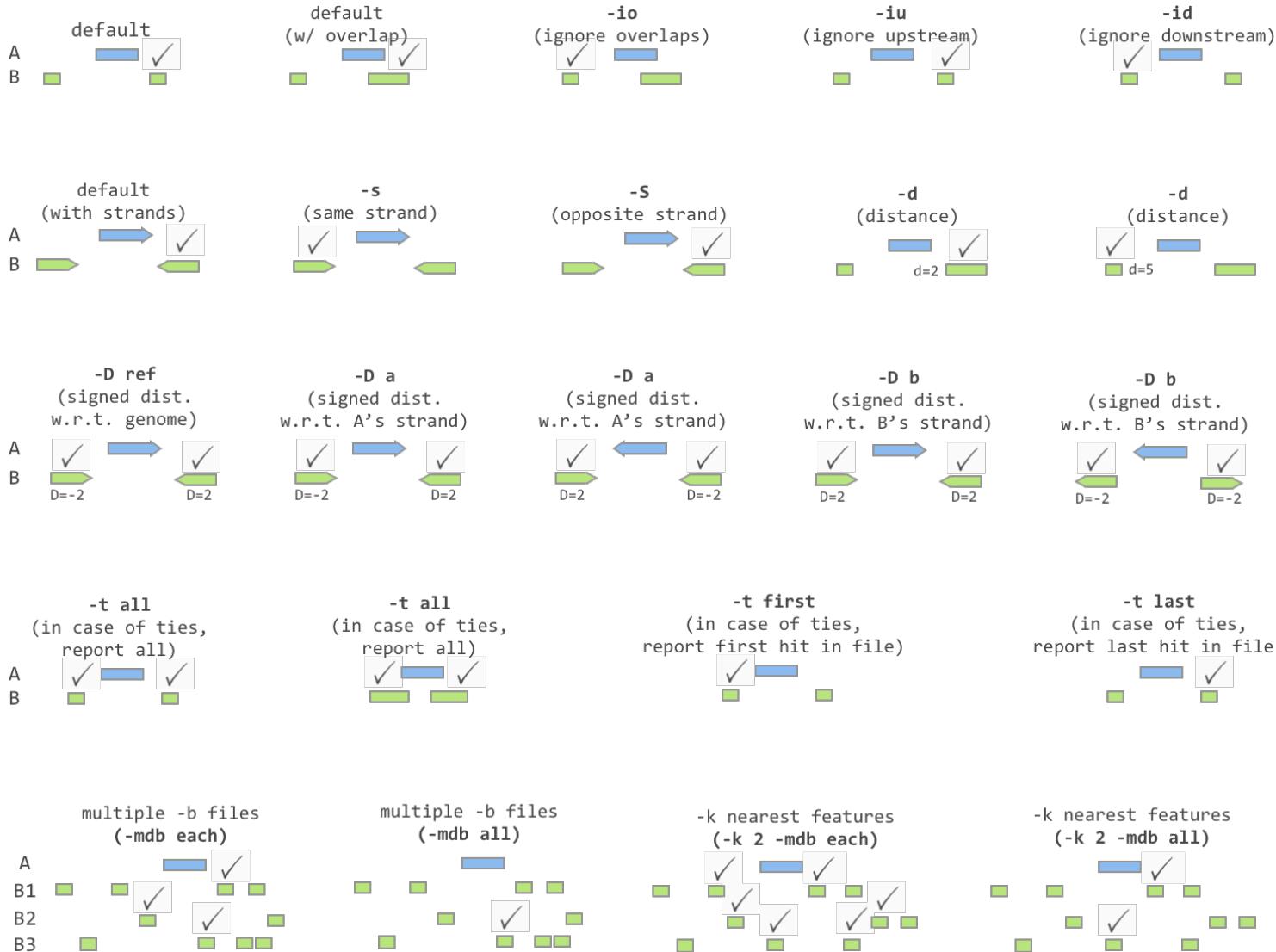
```
## genes.bed has: chrom, txStart, txEnd, name, num_exons, and strand
$ head -n4 genes.bed
chr1    134212701    134230065    Nuak2     8      +
chr1    134212701    134230065    Nuak2     7      +
chr1    33510655     33726603     Prim2,    14     -
chr1    25124320     25886552     Bai3,    31  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

```
## Identify promoter regions (2kbp upstream)
$ bedtools flank -i genes.bed -g mm9.chromsizes -l 2000 -r 0 -s > genes.2kb.promoters.bed
```

```
## Show promoter coordinates
$ head genes.2kb.promoters.bed
chr1    134210701    134212701    Nuak2     8      +
chr1    134210701    134212701    Nuak2     7      +
chr1    33726603     33728603     Prim2,    14     -
chr1    25886552     25888552     Bai3,    31     -
```

```
## Extract the sequences
$ bedtools getfasta -fi mm9.fa -bed genes.2kb.promoters.bed -fo genes.2kb.promoters.bed.fa
```

BEDTools Closest



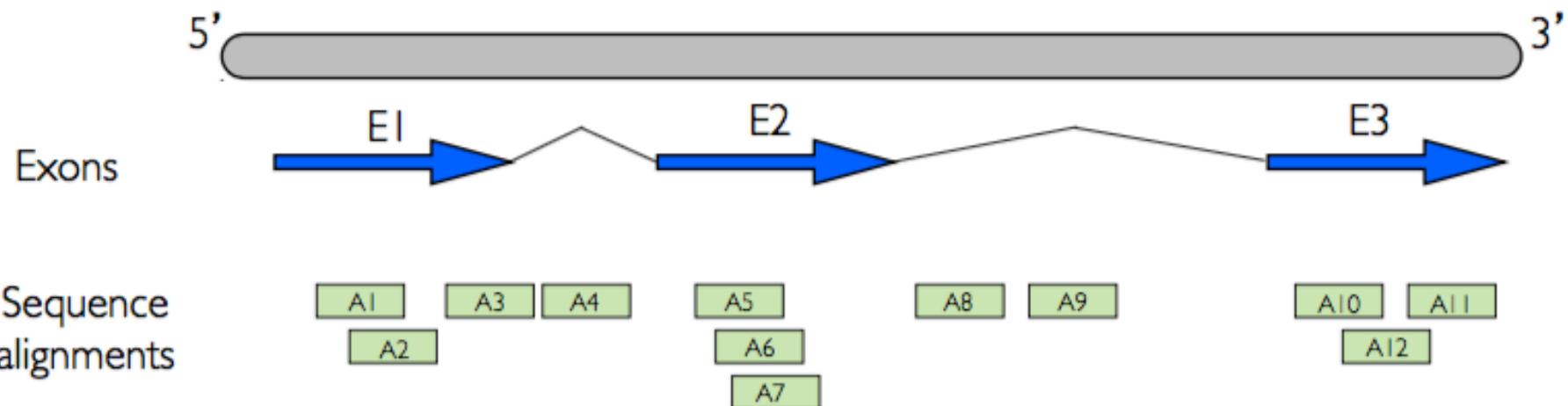
What is the gene closest to this SNP or this enhancer?

BEDTools commands

annotate	getfasta	overlap
bamtobed	groupby	pairstobed
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>

BEDTools Performance



How many reads are aligned to exonic sequences?

```
$ awk '{if ($3=="exon"){print}}' gencode.v21.annotation.gff3 | wc -l  
1162114
```

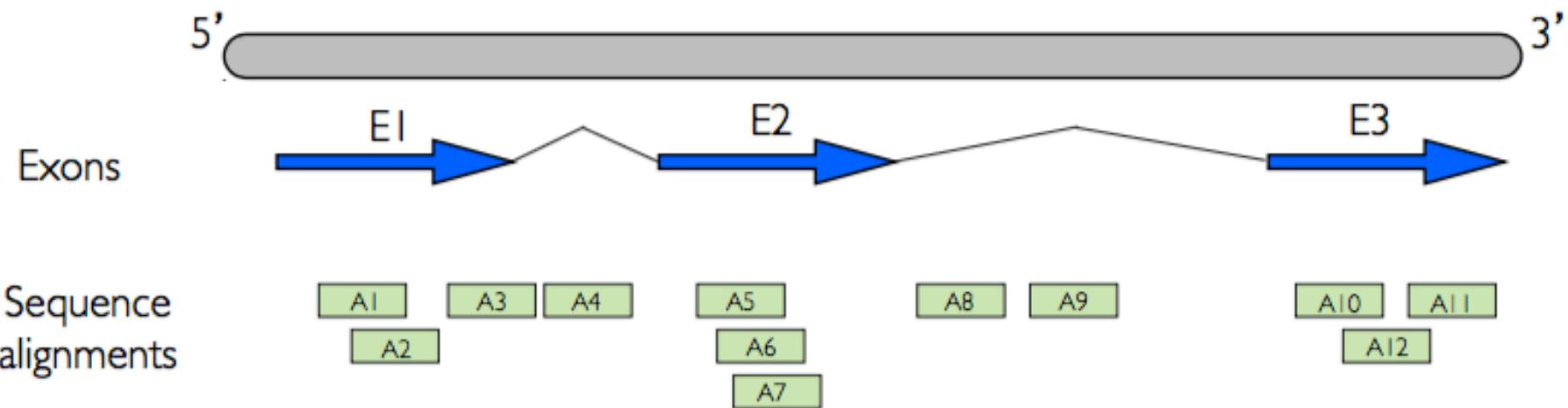
```
if ((read.start <= exon.end) && (read.end >= exon.start)) { print "in exon!"; }
```

How many comparison would a brute force approach take to scan a 30x dataset?

30x3Gb = 90Gbp / 100bp reads = 900M reads

900M reads x 1.1M exons = 990MM comparisons! ☹

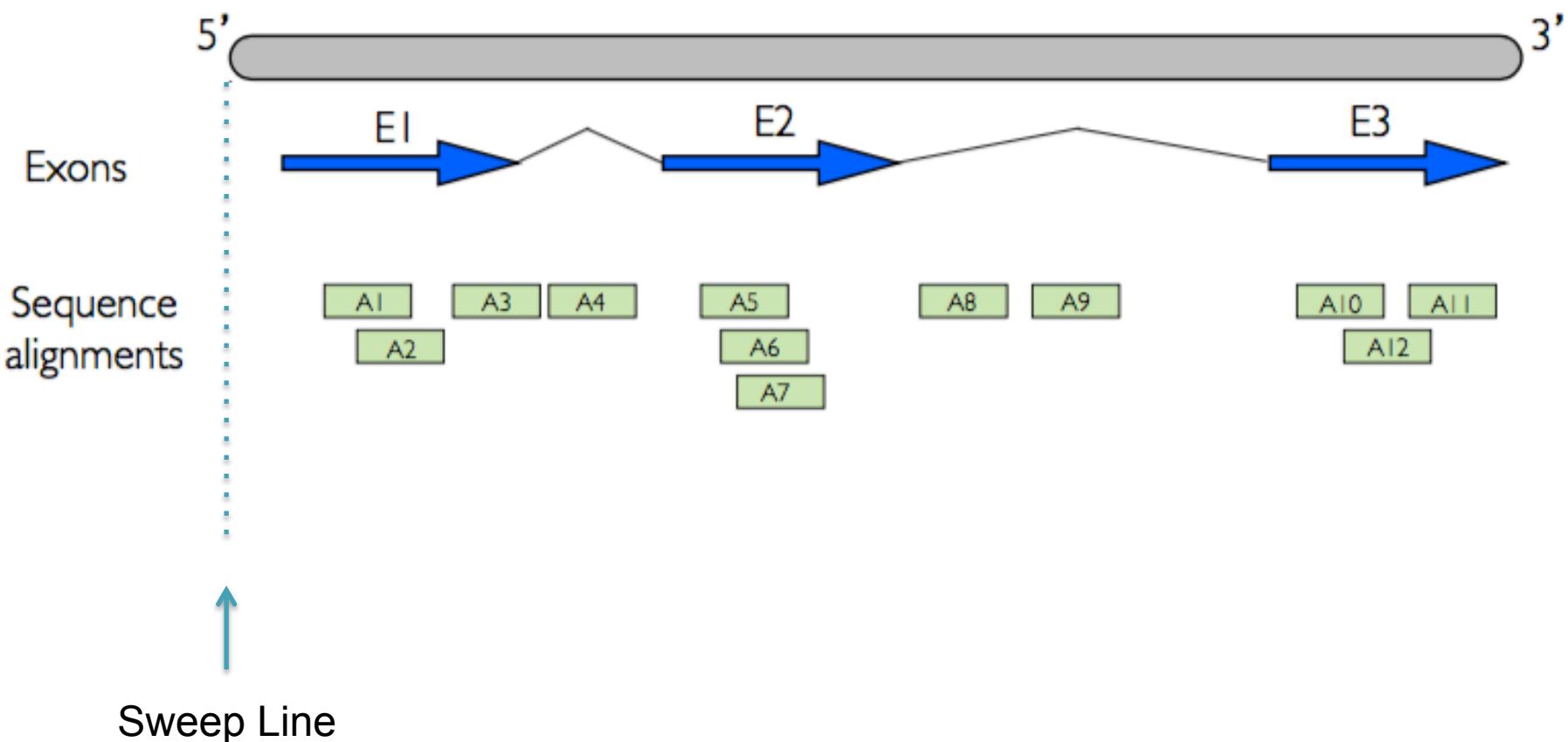
BEDTools Performance



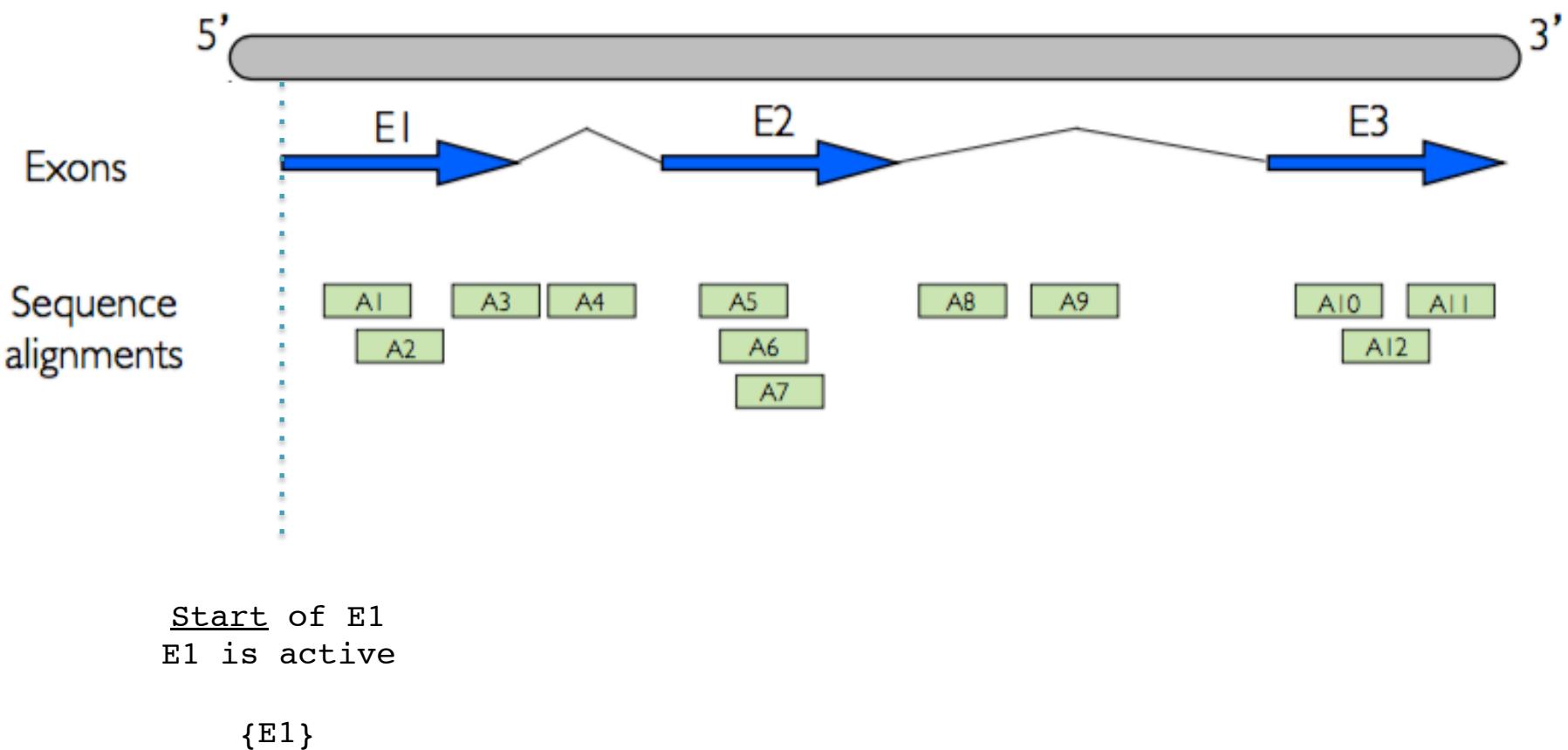
Assume datasets are sorted by chromosome and start position
samtools sort to sort alignments, unix sort to sort BED file

Any ideas?

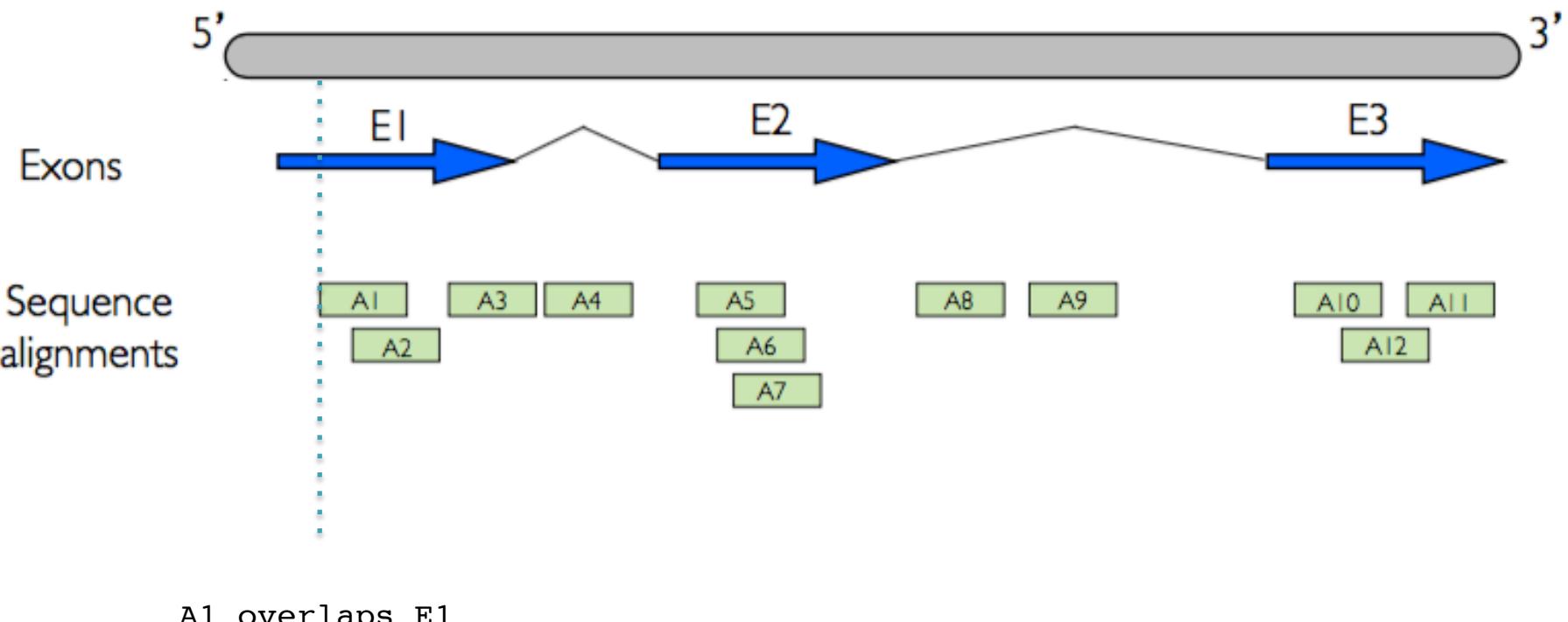
Plane Sweep to the Rescue!



Plane Sweep to the Rescue!



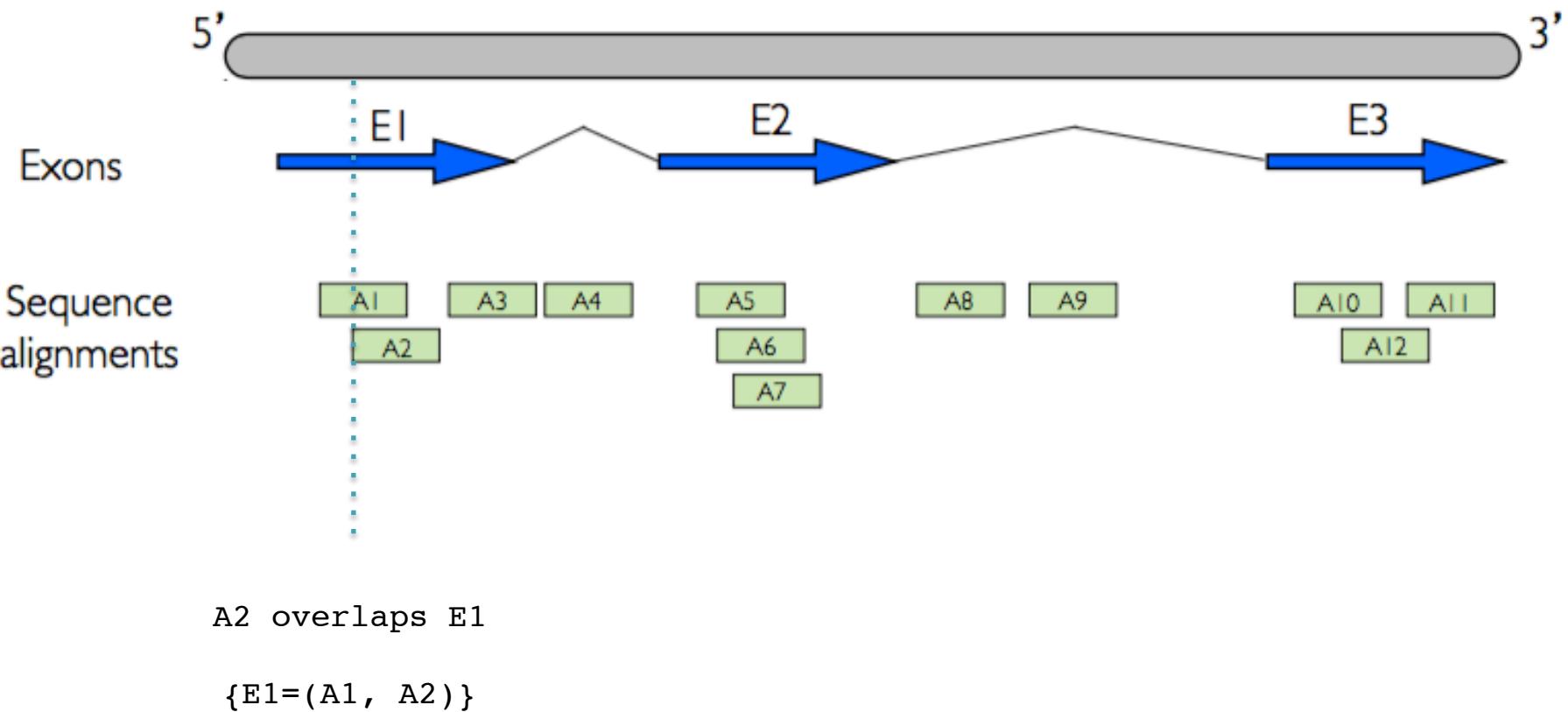
Plane Sweep to the Rescue!



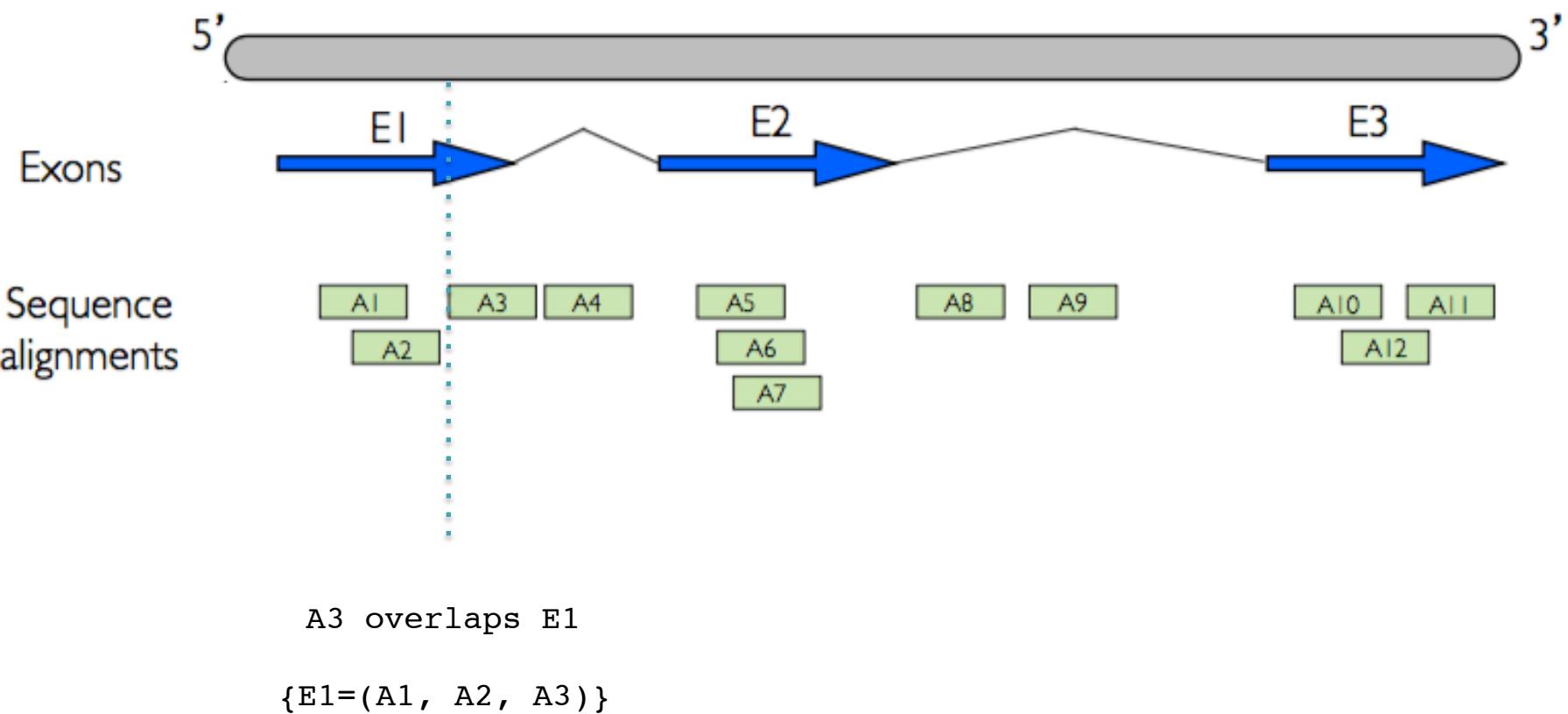
A1 overlaps E1

$\{E1=(A1)\}$

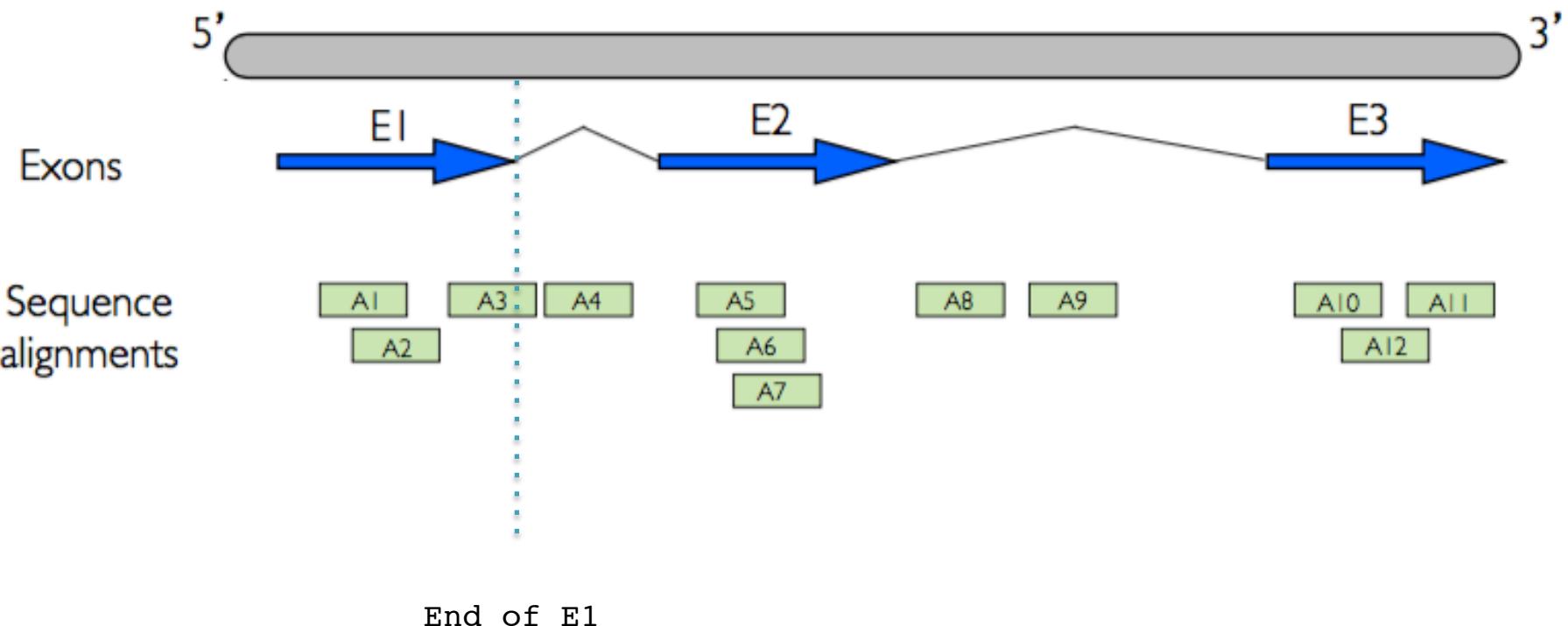
Plane Sweep to the Rescue!



Plane Sweep to the Rescue!



Plane Sweep to the Rescue!

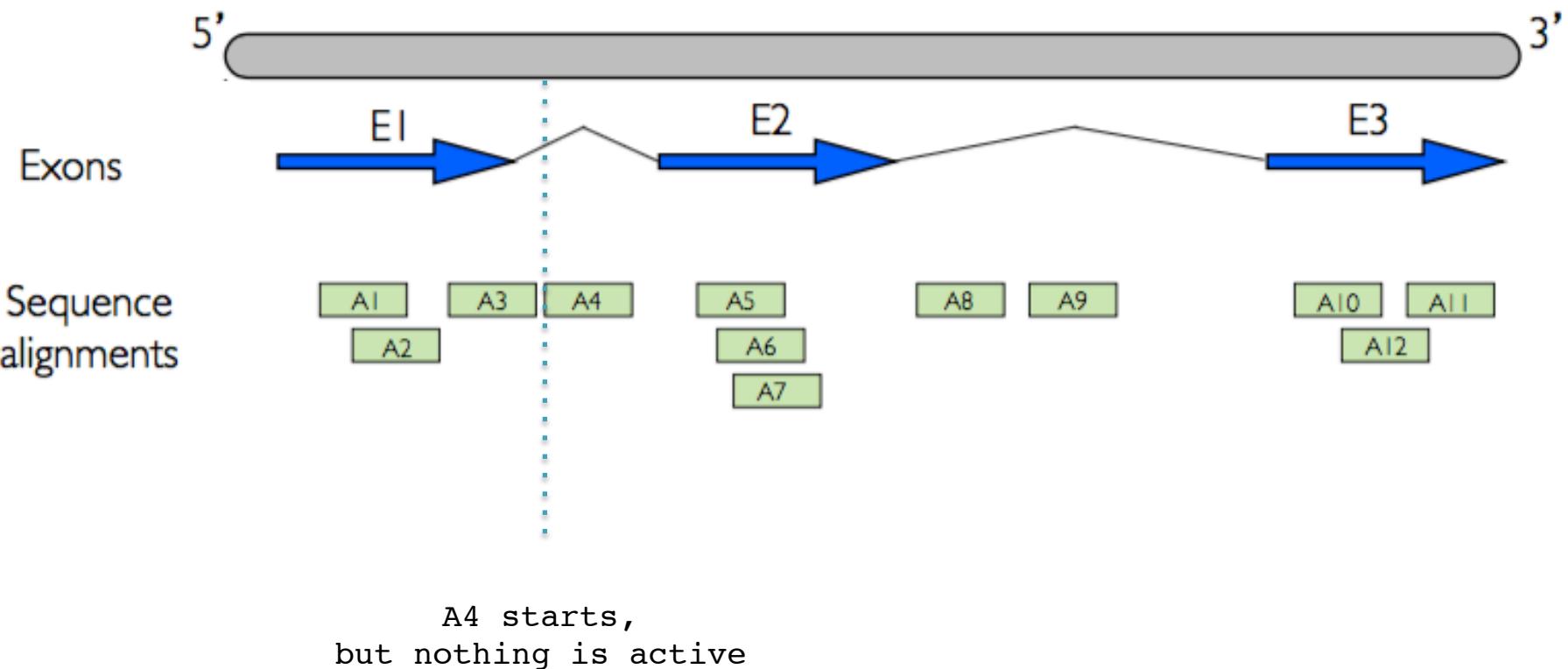


End of E1

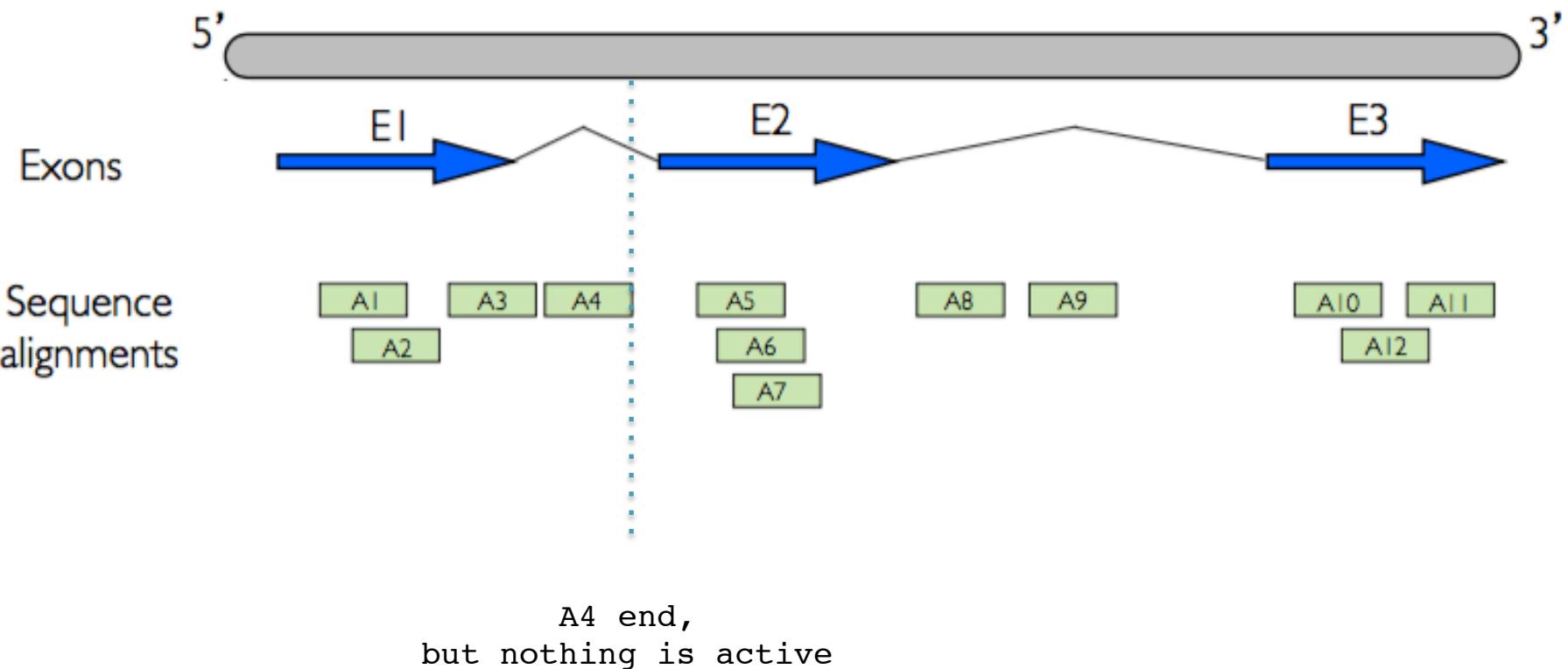
Report:

{E1=(A1, A2, A3)}

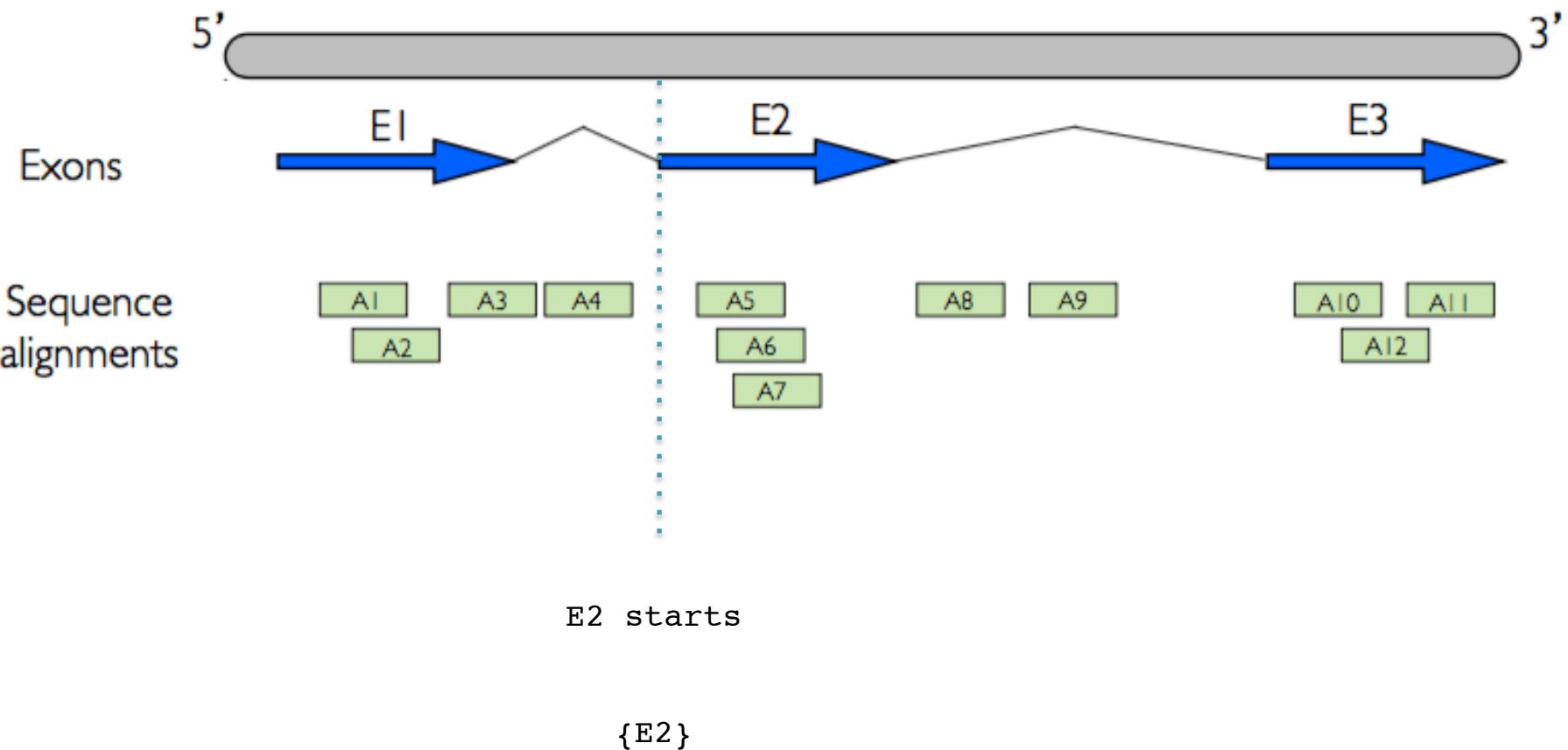
Plane Sweep to the Rescue!



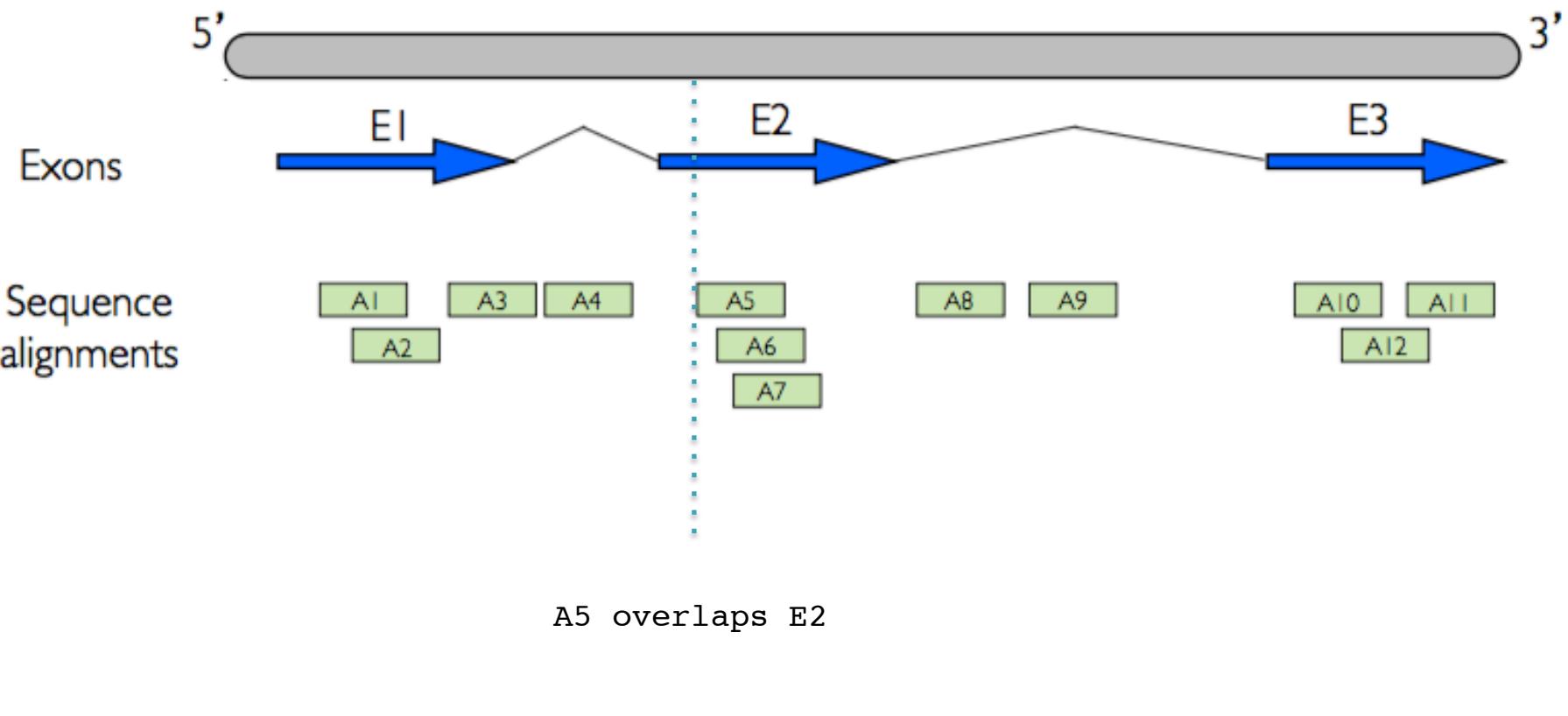
Plane Sweep to the Rescue!



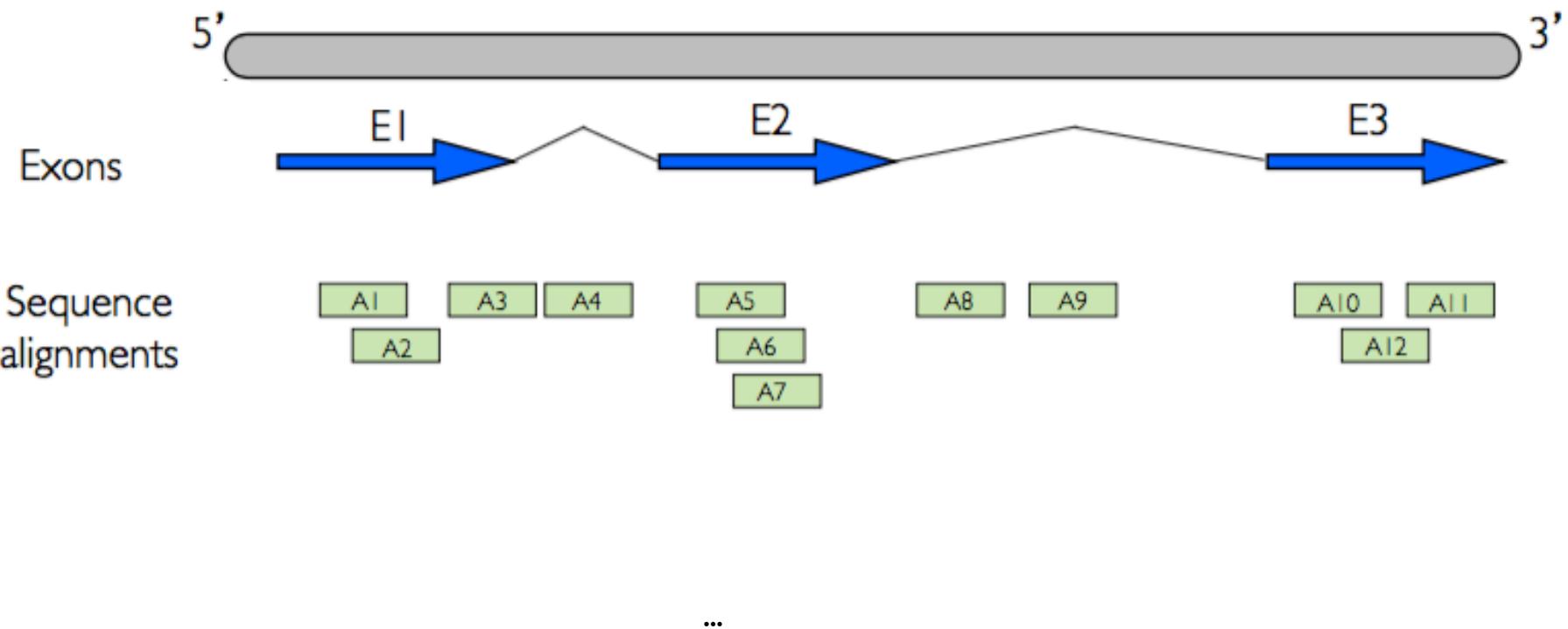
Plane Sweep to the Rescue!



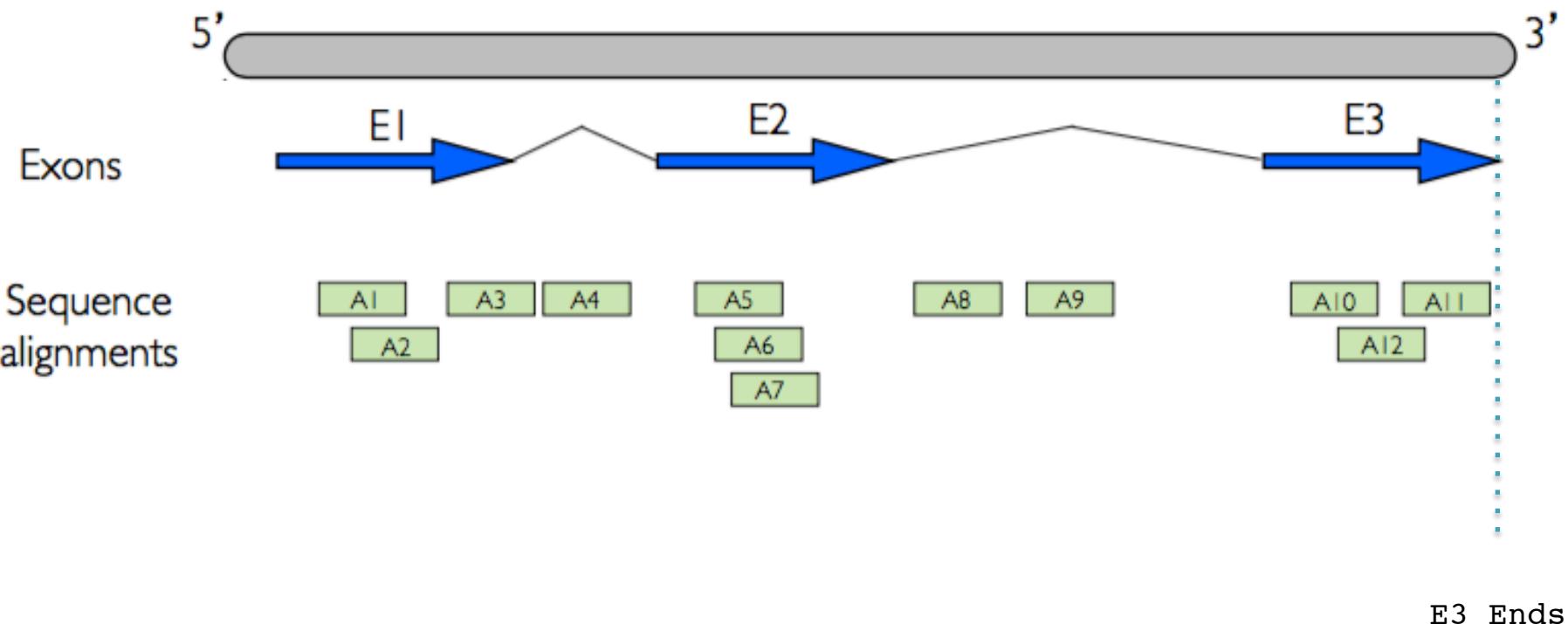
Plane Sweep to the Rescue!



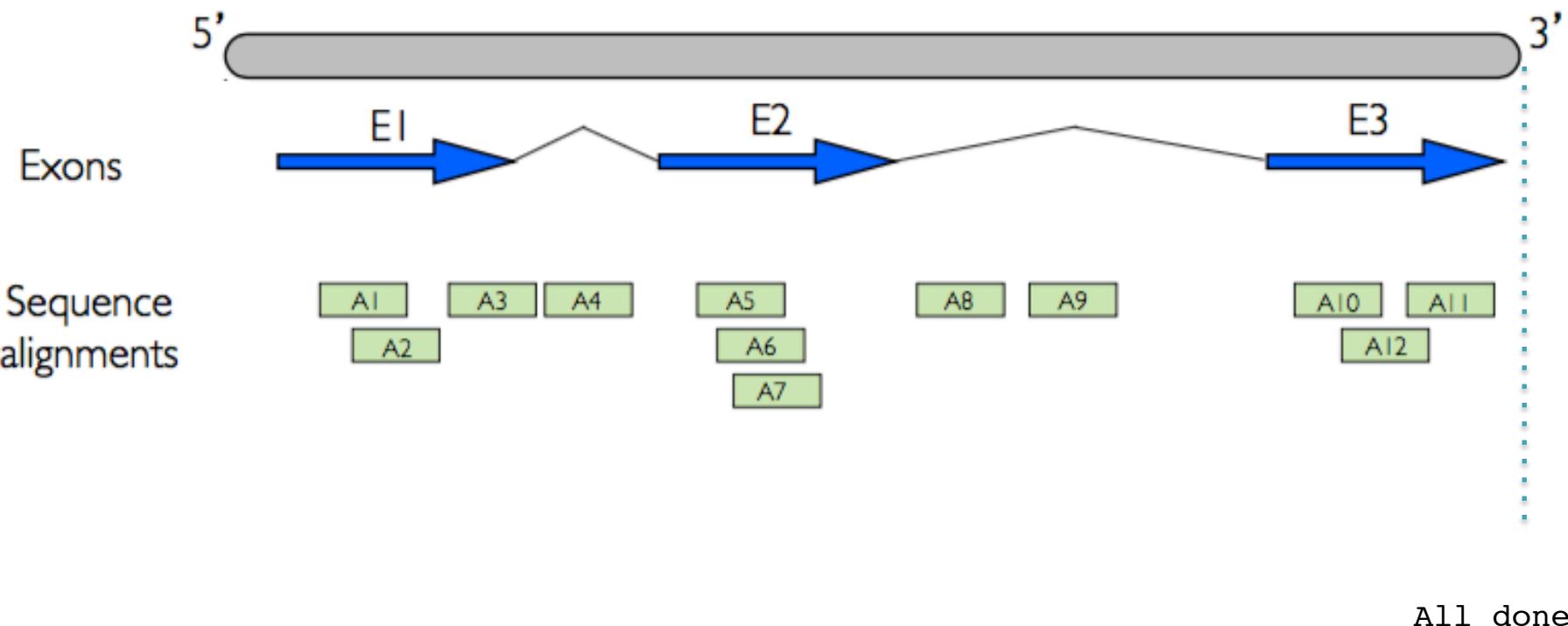
Plane Sweep to the Rescue!



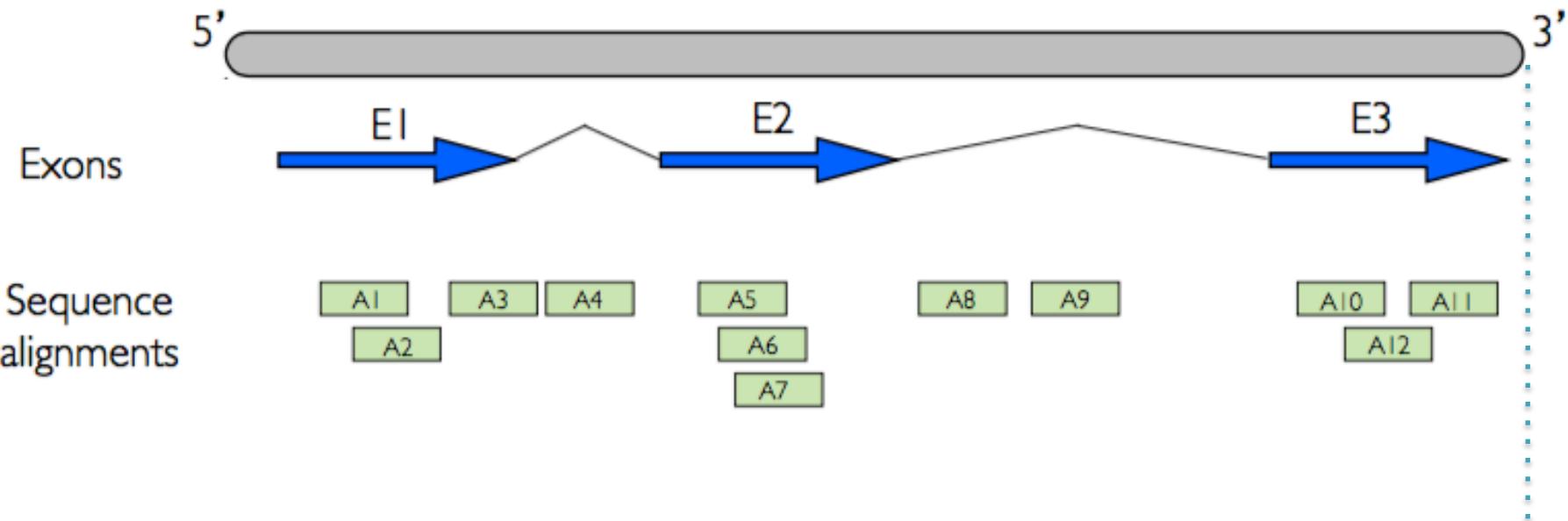
Plane Sweep to the Rescue!



Plane Sweep to the Rescue!



Plane Sweep to the Rescue!



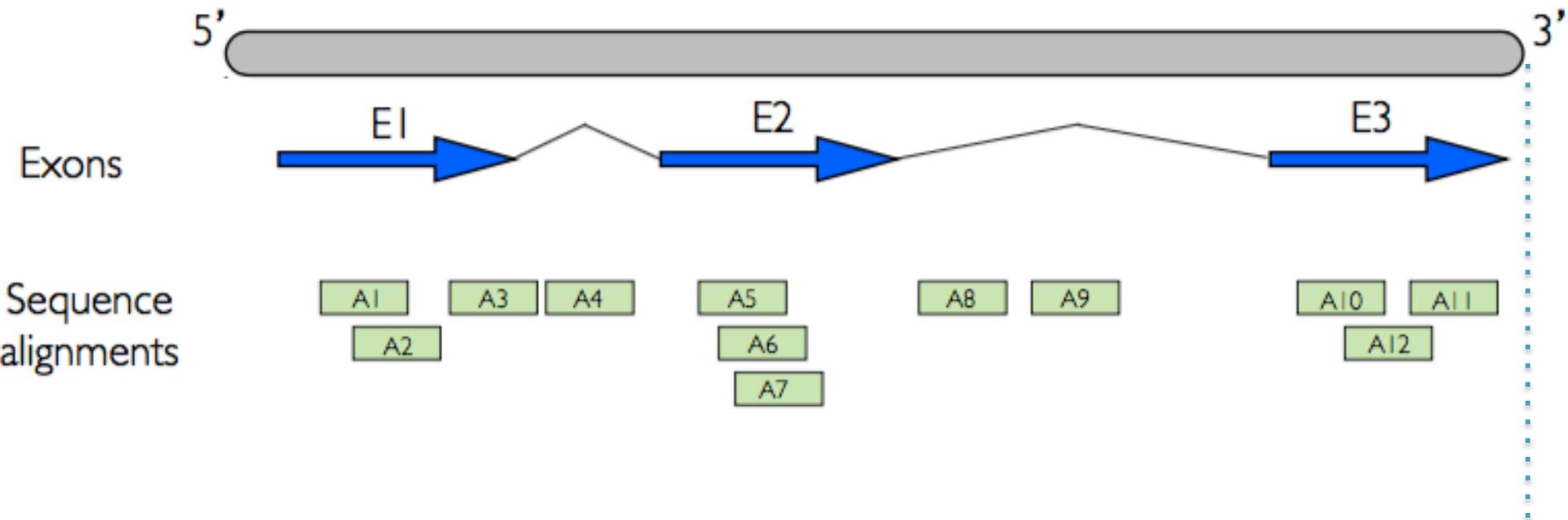
Final Results:

$$E1 = (A1, A2, A3)$$

$$E2 = (A5, A6, A7)$$

$$E3 = (A10, A11, A12)$$

Plane Sweep to the Rescue!



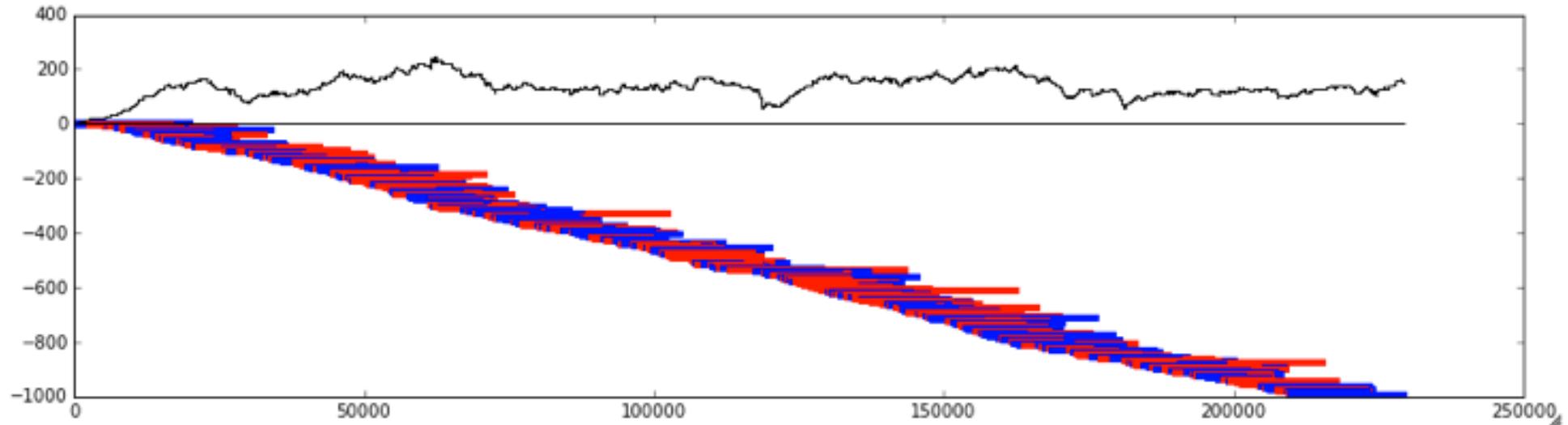
How many comparisons does the plane sweep algorithm make?

Each read is compared to the “active set”

Relatively few exons overlap: average ~1.1 active exons/position

Total comparisons: 900M reads * 1.1 “active exons/read” = 990M comparisons ☺

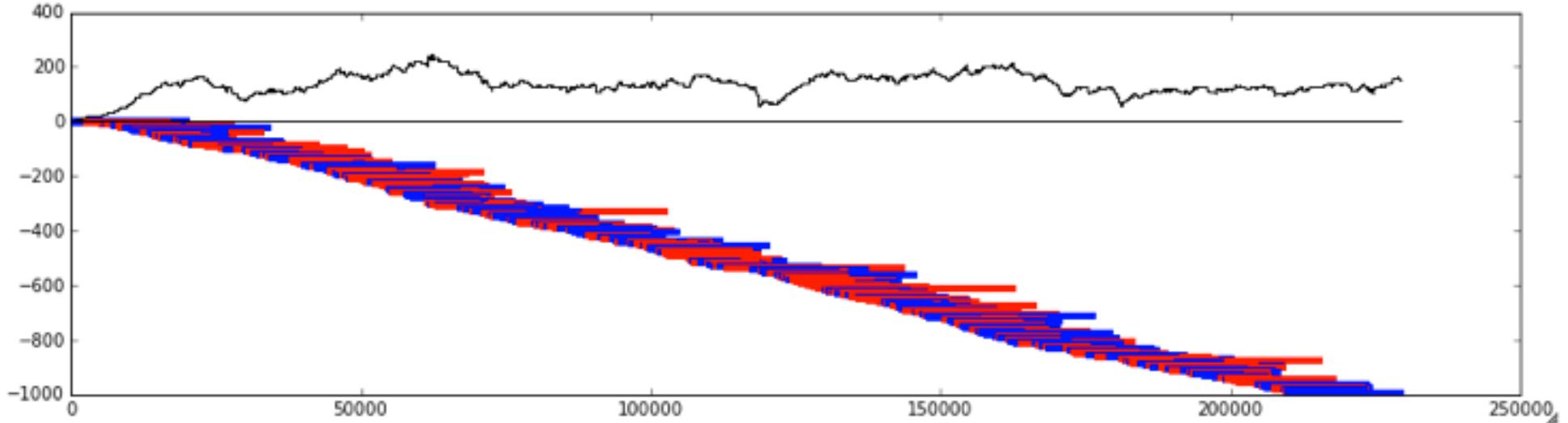
Coverage across the genome



```
$ 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]
```

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

```
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
==> Just like a plane sweep

```
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"

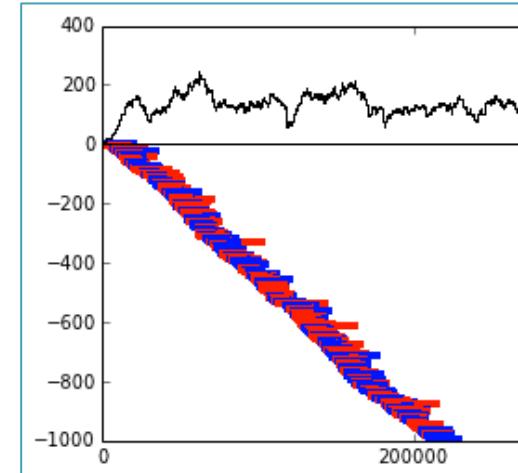
    plt.plot ([start,end], [-2*i, -2*i], lw=4, color=color)

## 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



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

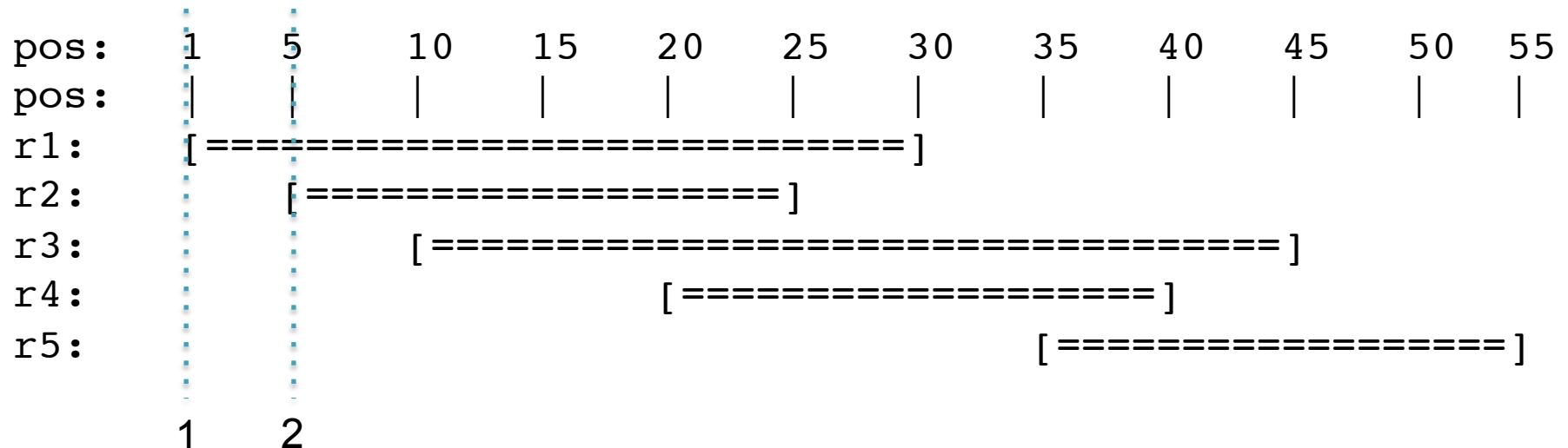
	1	5	10	15	20	25	30	35	40	45	50	55
pos:
pos:												
r1:	[=====	=====	=====	=====	=====	=====	=====	=====	=====	=====]
r2:					=====	=====	=====	=====	=====	=====	=====]
r3:						=====	=====	=====	=====	=====	=====]
r4:							=====	=====	=====	=====	=====]
r5:								=====	=====	=====	=====]
	1											

arrive at r1 [1,30]:

active set is empty; add to active set: 30

output (1,1)

Plane Sweep



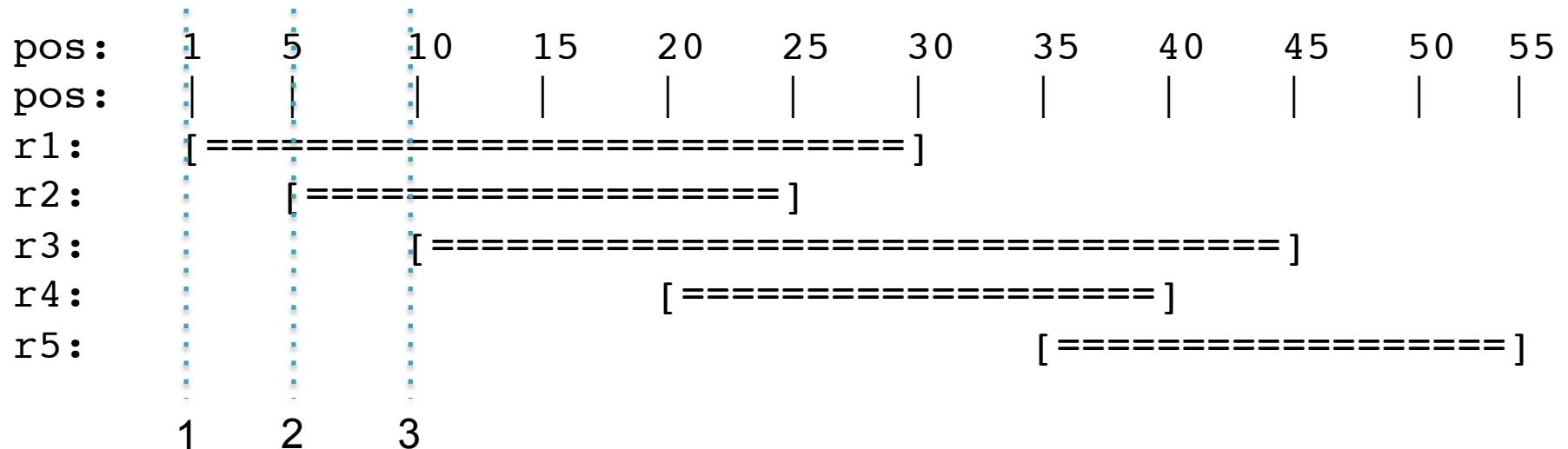
arrive at r1 [1,30]:

*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 $r_1 [1,30]$:

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

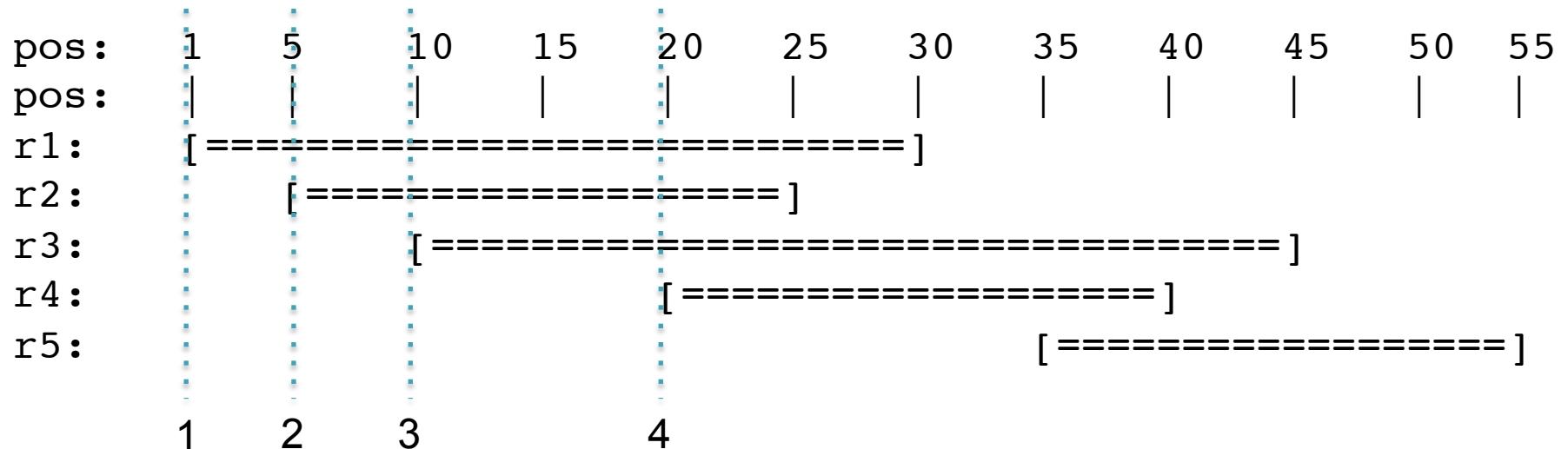
arrive at $r_2 [5,25]$:

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

arrive at $r_3 [10,45]$:

$10 < 25$: 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

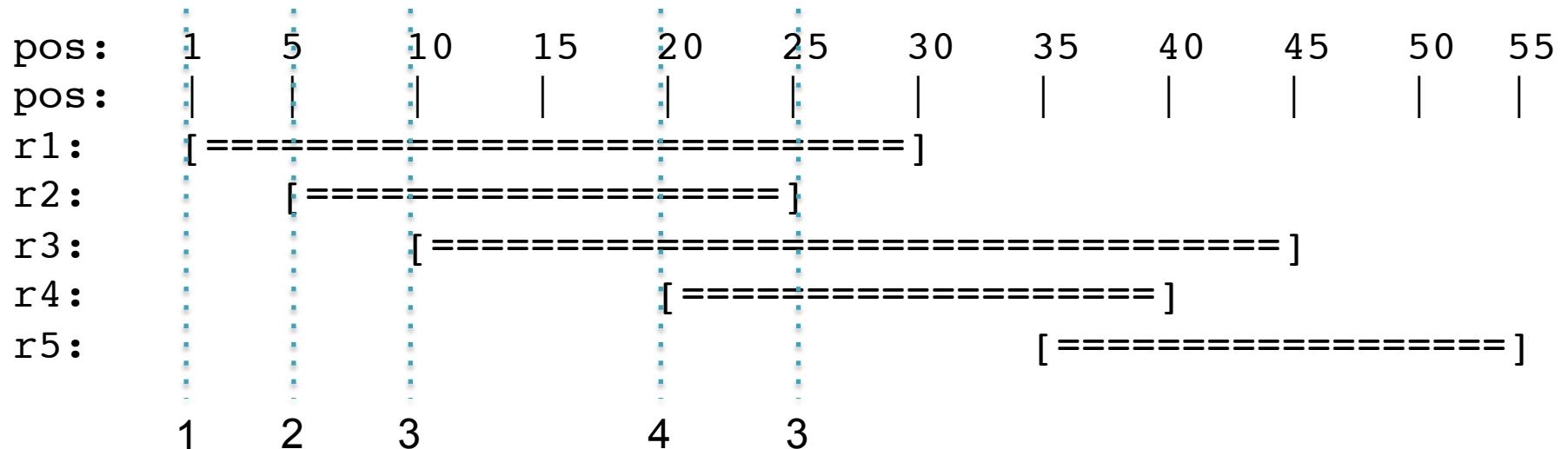
output (10, 3)

arrive at r4 [20,40]:

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

output (20, 4)

Plane Sweep

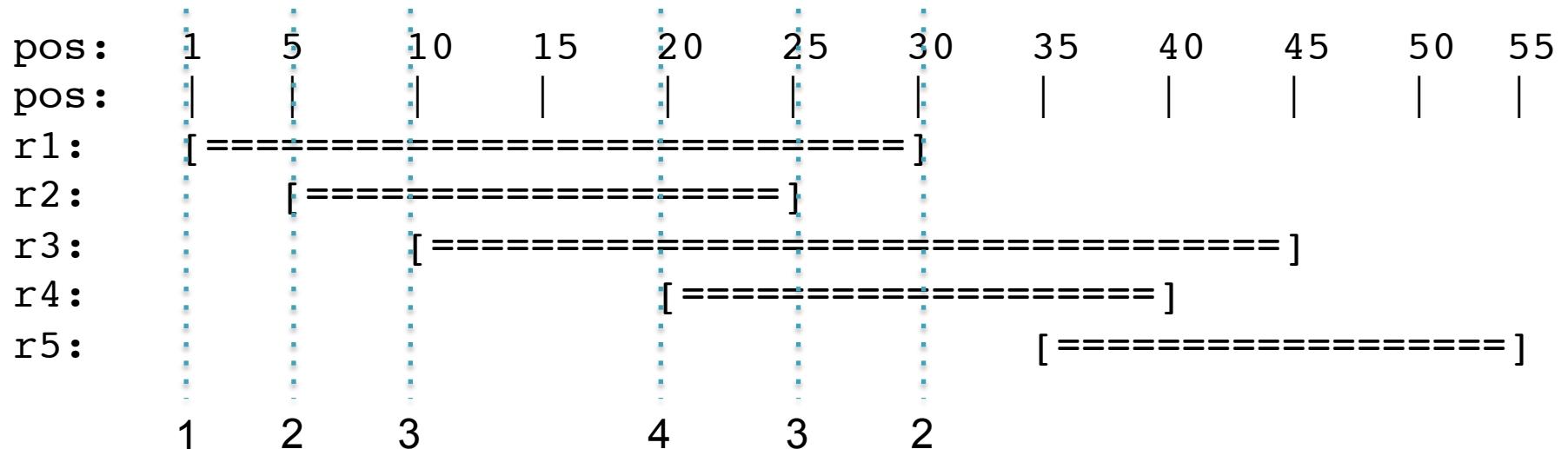


arrive at r5[35,55]:

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

output (25, 3)

Plane Sweep

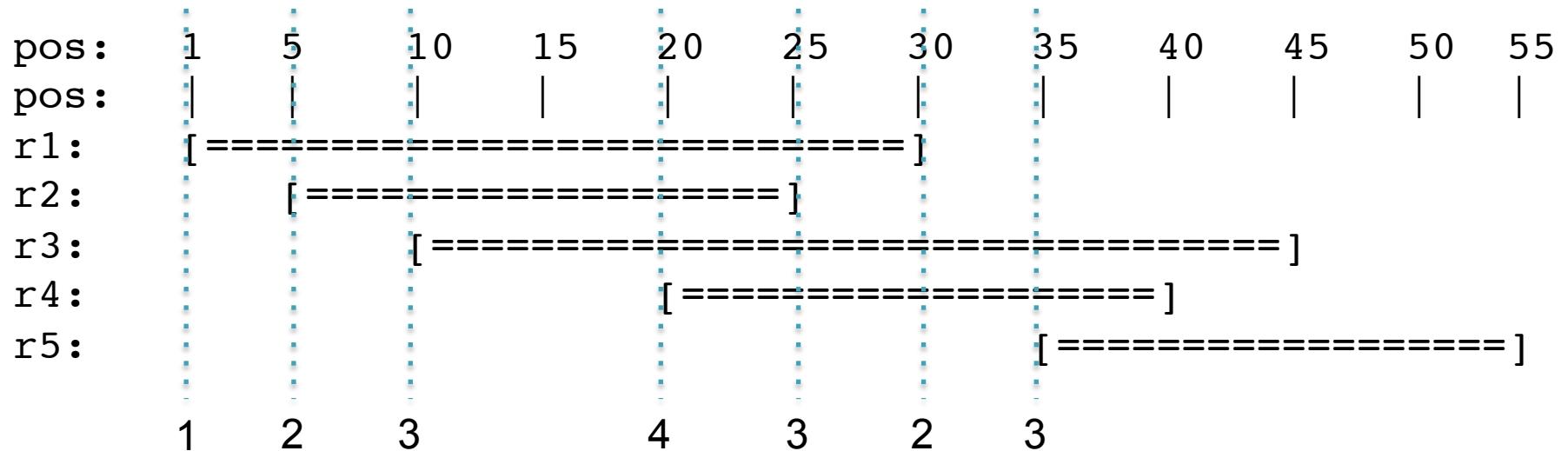


arrive at r5[35,55]:

*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



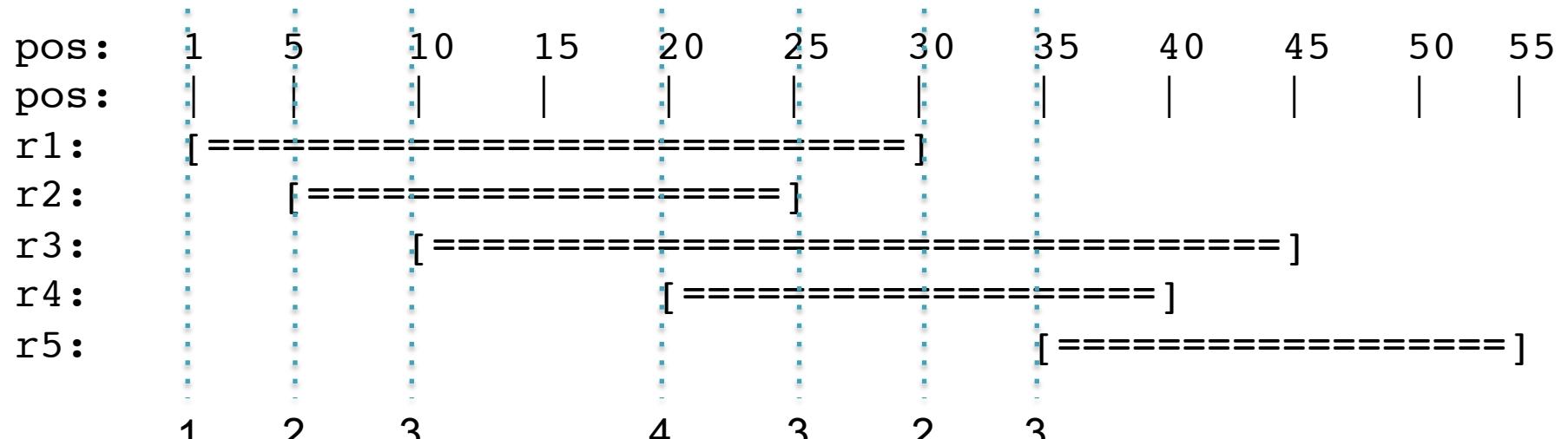
arrive at r5[35,55]:

*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)*

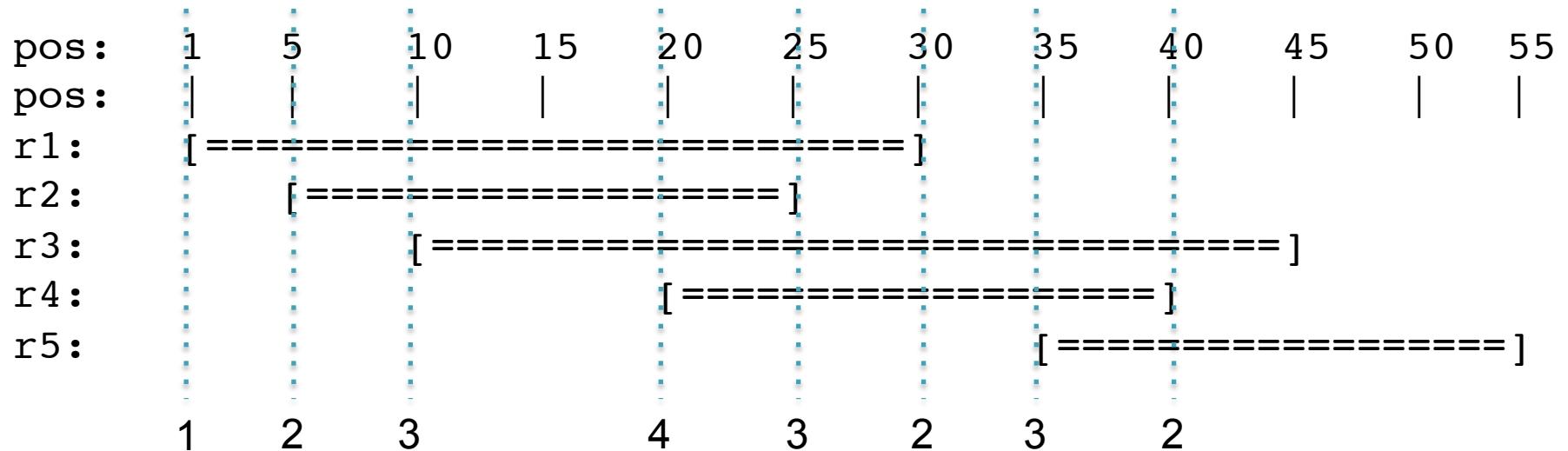
*35 < 40: add to active set: 40, 45, 55
output (35, 3)*

Plane Sweep



Flush:

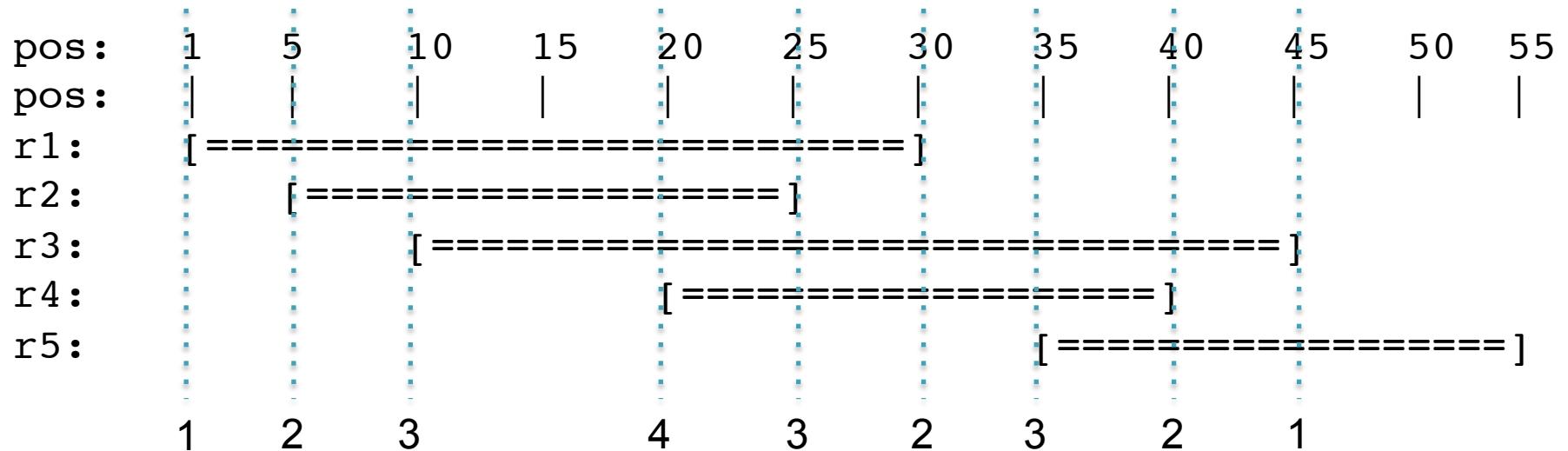
Plane Sweep



Flush:

*step down at 40; active set: 45, 55
output (40, 2)*

Plane Sweep

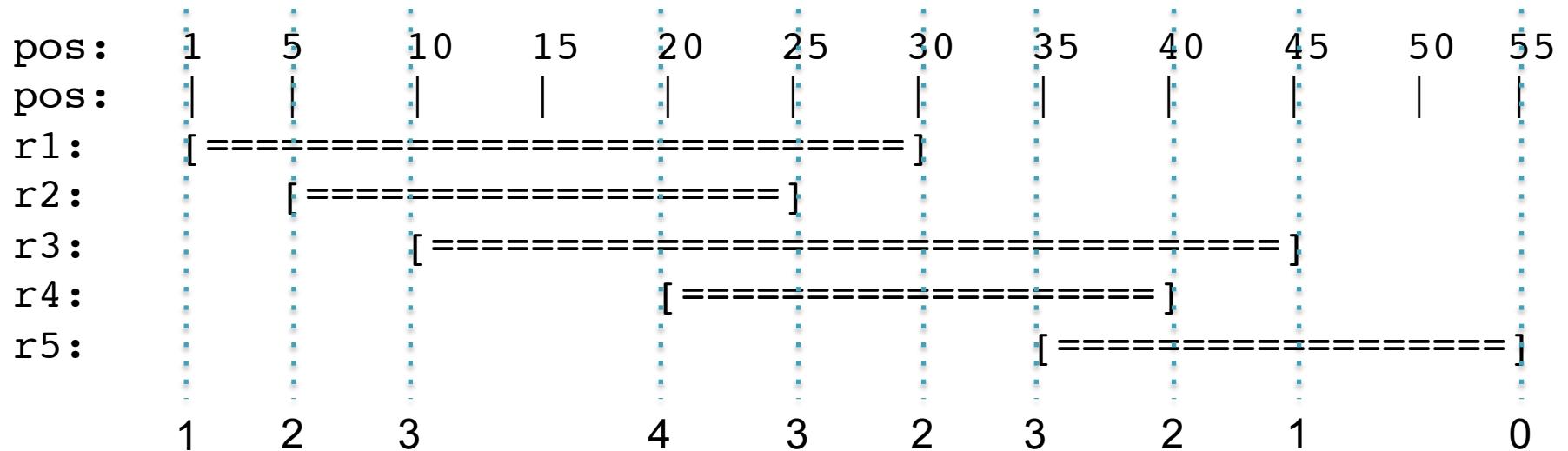


Flush:

*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)
            deltaxcovplane.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
    deltaxcovplane.append((startpos, len(planelist)))

## Flush any remaining end positions
while (len(planelist) > 0):
    oldend = planelist.pop(0)
    deltaxcovplane.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?

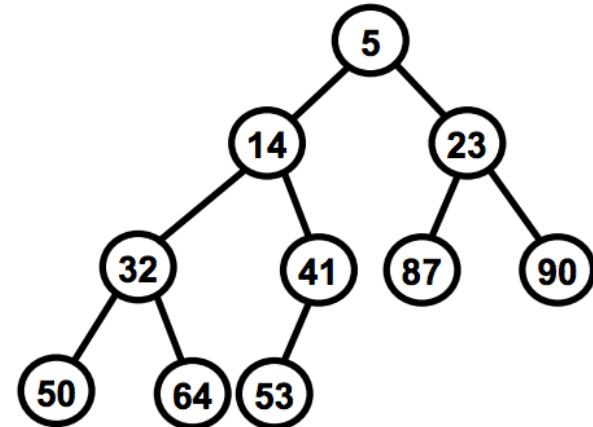
Beginning list-based plane sweep over 1873 reads

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 Software Foundation's documentation for the `heapq` module. The title bar reads "8.4. heapq — Heap queue alg". 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 "Basic Examples", "Priority Queue Implementation Notes", and "Theory". The main content area describes the heap queue algorithm, its implementation using zero-based indexing, and provides examples for functions like `heappush`, `heappop`, `heappushpop`, `heapify`, `heappreplace`, `merge`, and `nlargest`.

Table Of Contents

- 8.4. `heapq` — Heap queue algorithm
 - 8.4.1. Basic Examples
 - 8.4.2. Priority Queue Implementation Notes
 - 8.4.3. Theory

Previous topic

[8.3. `collections` — High-performance container datatypes](#)

Next topic

[8.5. `bisect` — Array bisection algorithm](#)

This Page

[Report a Bug](#) [Show Source](#)

Quick search

Enter search terms or a module, class or function name. Go

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.heappreplace(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 dataset 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

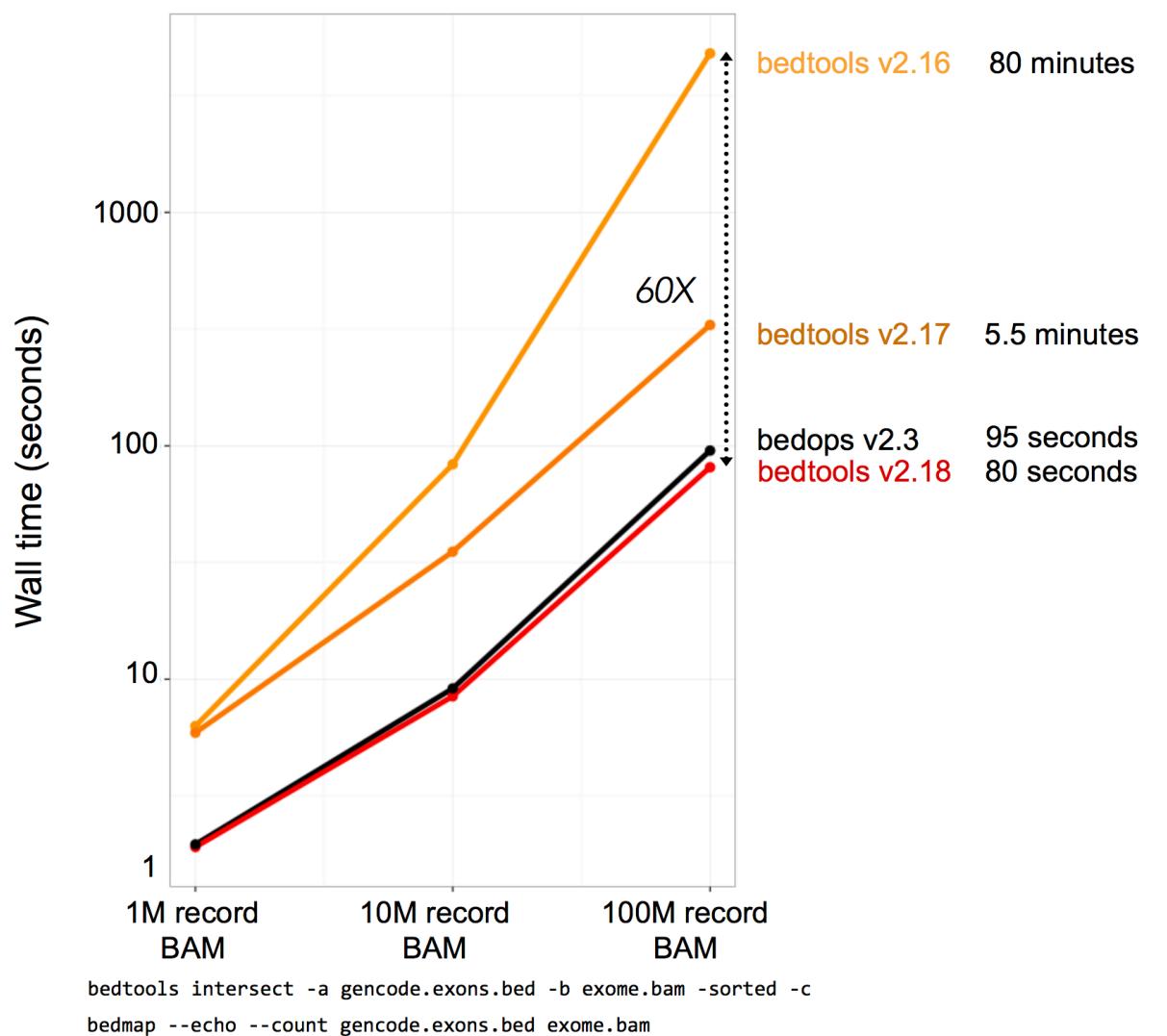
heapq.heappop() removes from heap

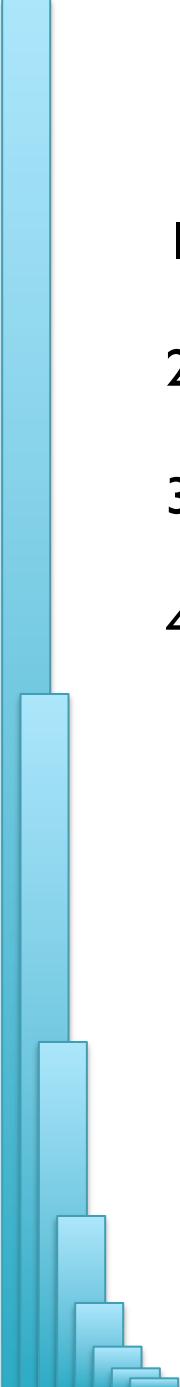
heapq.heappush() adds to heap

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)!

BEDTools Performance





Next Steps

1. See Lecture Notes for Full Details
2. Review Bedtools docs: <http://bedtools.readthedocs.io/>
3. Get ready for assignment 2
4. Check out the course webpage



Welcome to Applied Comparative Genomics

<https://github.com/schatzlab/appliedgenomics>

Questions?