Genome Arithmetic

Michael Schatz

Feb 17, 2020

Lecture 7: Applied Comparative Genomics

Assignment 2: Genome Assembly
Due Wednesday Feb 12 @ 11:59pm

1. ***Setup Docker/Ubuntu***
2. ***Initialize Tools***
3. ***Download Reference Genome & Reads***
4. ***Decode the secret message***
    1. *Estimate coverage, check read quality*
    2. *Check kmer distribution*
    3. *Assemble the reads with spades*
    4. *Align to reference with MUMmer*
    5. *Extract foreign sequence*
    6. *dna-encode.pl -d*

https://github.com/schatzlab/appliedgenomics2020/blob/master/assignments/assignment2/README.md

Assignment 3: Due Wed Feb 19

Assignment 3: Coverage, Genome Assembly, and Variant Calling

Assignment Date: Wednesday, Feb. 12, 2020
Due Date: Wednesday, Feb. 19, 2020 @ 11:59pm

Some of the tools you will need to use only run in a linux or mac environment. If you do not have access to a linux/mac machine, download and install a virtual machine or ubuntu instance following the directions here: https://github.com/schatzlab/appliedgenomics2018/blob/master/assignments/virtualbox.md

Alternatively, you might also want to try out this docker instance that has these tools preinstalled: https://github.com/mschatz/wga-essentials

Question 1. Coverage simulator [10 pts]

- Q1a. How many 100bp reads are needed to sequence a 1Mbp genome to 5x coverage?
- Q1b. In the language of your choice, simulate sequencing 5x coverage of a 1Mbp genome and plot the histogram of coverage. Note you do not need to actually output the sequences of the reads, you can just randomly sample positions in the genome and record the coverage. You do not need to consider the strand of each read. The start position of each read should have a uniform random probability at each possible starting position (1 through 999,900). You can record the coverage in an array of 1M positions. Overlay the histogram with a Poisson distribution with lambda=5
- Q1c. Using the histogram from 1b, how much of the genome has not been sequenced (has 0x coverage). How well does this match Poisson expectations?
- Q1d. Now repeat the analysis with 15x coverage: 1. simulate the appropriate number of reads, 2. make a histogram, 3. overlay a Poisson distribution with lambda=15, 4. compute the number of bases with 0x coverage, and 5. evaluated how well it matches the Poisson expectation.

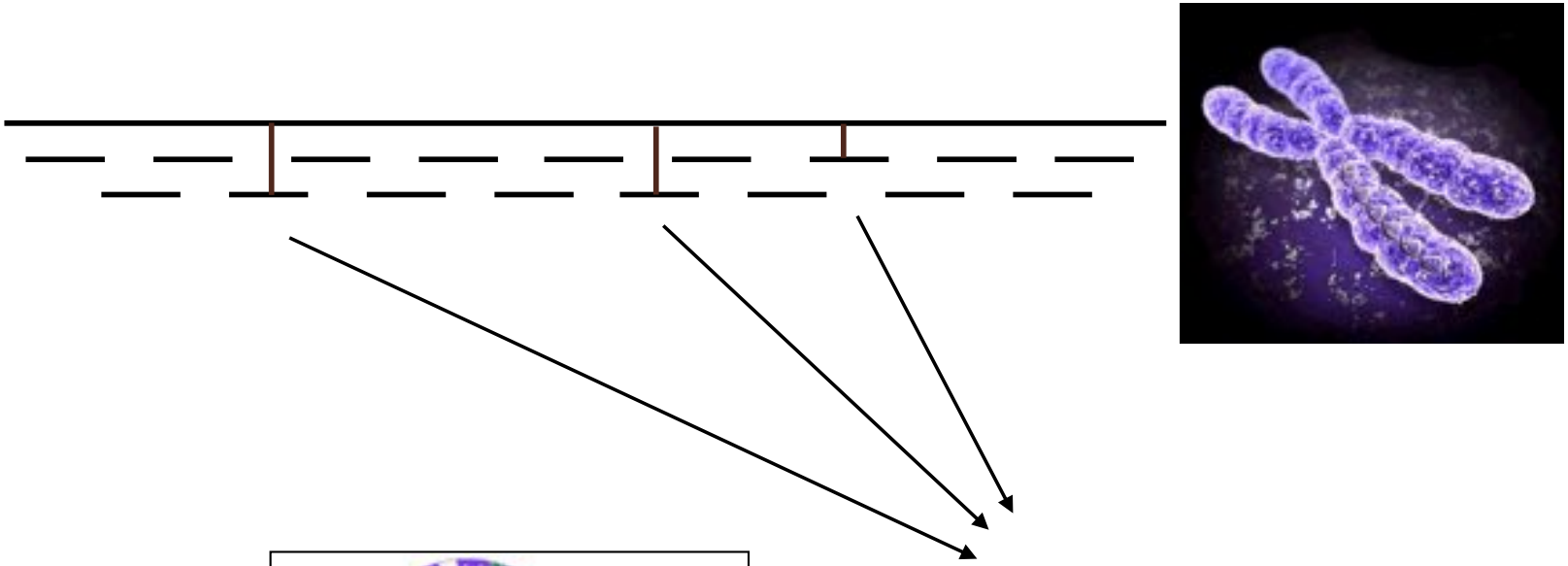
Question 2. de Bruijn Graph construction [10 pts]

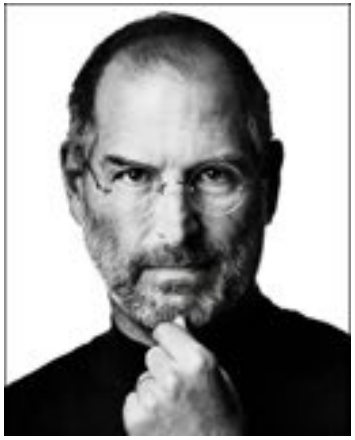
- Q2a. Draw (by hand or by code) the de Bruijn graph for the following reads using k=3 (assume all reads are from the forward strand, no sequencing errors, complete coverage of the genome)

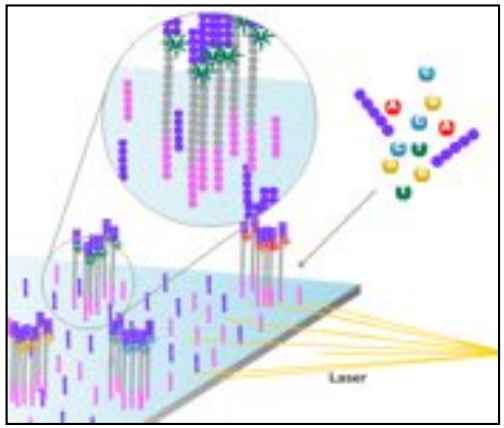
```
ATTCA
ATTGA
CATTG
CTTAT
GATTG
TATTT
```

Personal Genomics

How does your genome compare to the reference?





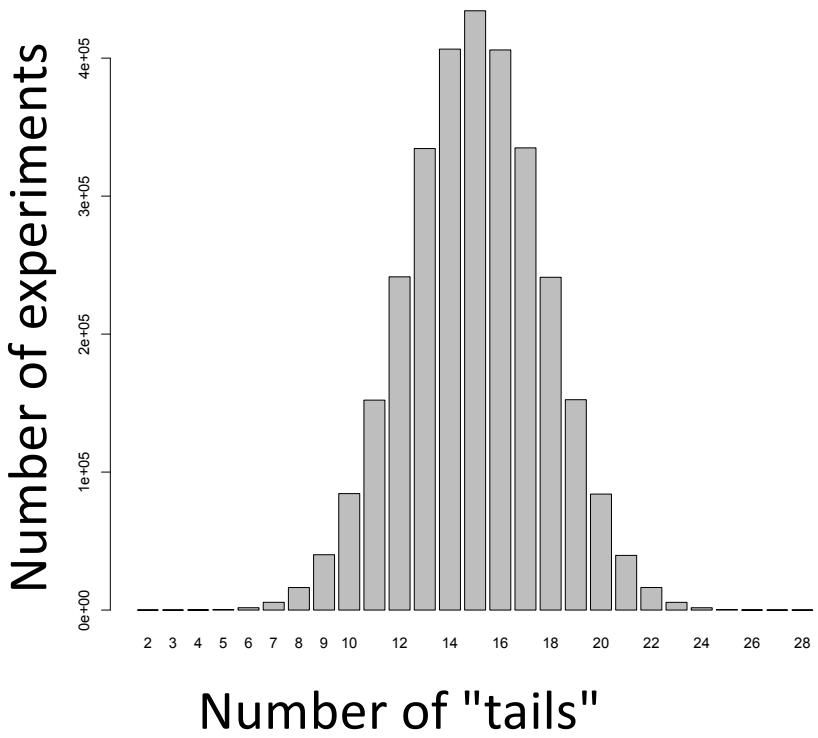


Heart Disease

Cancer

Creates magical technology

So, with 30 tosses (reads), we are much more likely to see an even mix of alternate and reference alleles at a heterozygous locus in a genome



This is why <u>at least</u> a "30X" (30 fold sequence coverage) genome is recommended: it confers sufficient power to distinguish heterozygous alleles and from mere sequencing errors

P(3/30 het) <?> P(3/30 err)

Thinking about allele sampling with the binomial distribution

The **binomial distribution** with parameters *n* and *p* is the discrete probability distribution of the number of successes in a sequence of *n* independent yes (e.g., "heads" or "reference allele") or no (e.g., "tails", or "alternate allele") experiments, each of which yields success with probability *p*.

The probability of getting exactly k successes in n trials is given by the probability mass function:

$$\Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

What is the probability of seeing k=1 tails in n=3 flips of a fair coin with the probability of a tail (p) = 0.5?

3 choose 1 = 3; 0.5^1 = 0.5; $(1-0.5)^{(3-1)}$ = 0.25. So…. 3*0.5*0.25 = **0.375**

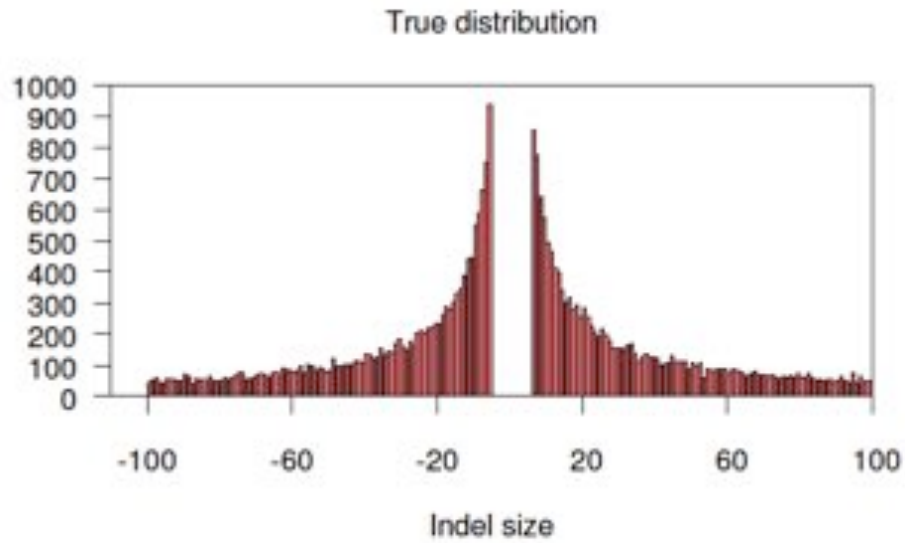
In R, the function would be: `dbinom(1, size=3, prob=0.5)`

Variation Detection Complexity

SNPs + Short Indels

High precision and sensitivity

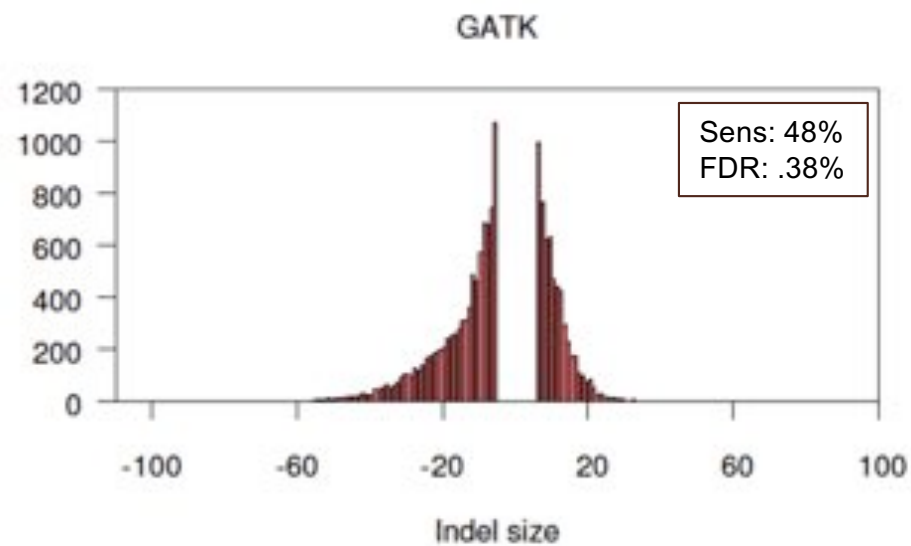
```
..TTTAGAATAG-CGAGTGC...
      |||||||  ||||
     AGAATAGGCGAG
```



"Long" Indels (>5bp)

Reduced precision and sensitivity

```
..TTTAG--------AGTGC...
  |||||        |||||
  TTTAGAATAGGC |||||
        ATAGGCGAGTGC
```



Analysis confounded by sequencing errors, localized repeats, allele biases, and mismapped reads

Scalpel: Haplotype Microassembly

DNA sequence **micro-assembly** pipeline for accurate detection and validation of *de novo* mutations (SNPs, indels) within exome-capture data.

Features

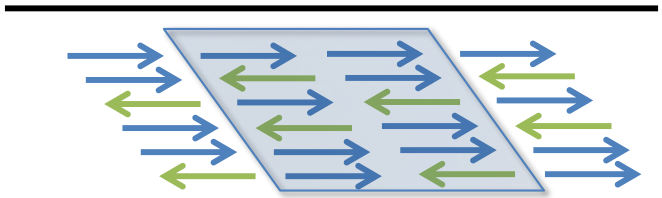
1. Combine mapping and assembly
2. Exhaustive search of haplotypes
3. De novo mutations

NRXN1 *de novo* SNP
(auSSC12501 chr2:50724605)

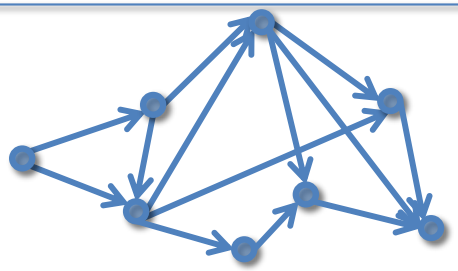
Accurate de novo and transmitted indel detection in exome-capture data using microassembly.
Narzisi et al. (2014) *Nature Methods*. doi:10.1038/nmeth.3069

Scalpel Algorithm

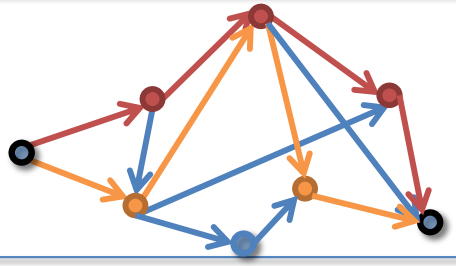
Extract reads mapping within the exon including (1) well-mapped reads, (2) soft-clipped reads, and (3) anchored pairs



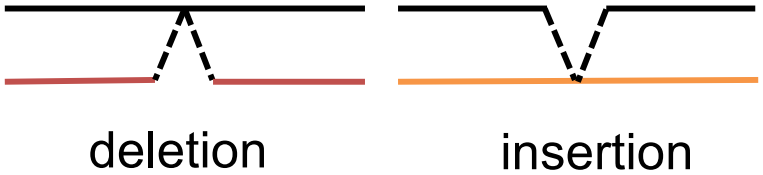
Decompose reads into overlapping *k*-mers and construct de Bruijn graph from the reads



Find end-to-end haplotype paths spanning the region



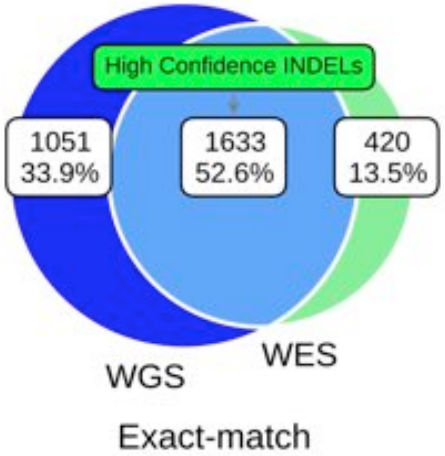
Align assembled sequences to reference to detect mutations



Refined indel analysis

Examine sources of indel errors

- Experimental validation of indels called from 30x whole genome vs. 110x whole exome of the same sample
- Most of the errors due to short microsatellite errors introduced during exome capture, also misses most long indels
- Recommend WGS for indel analysis instead

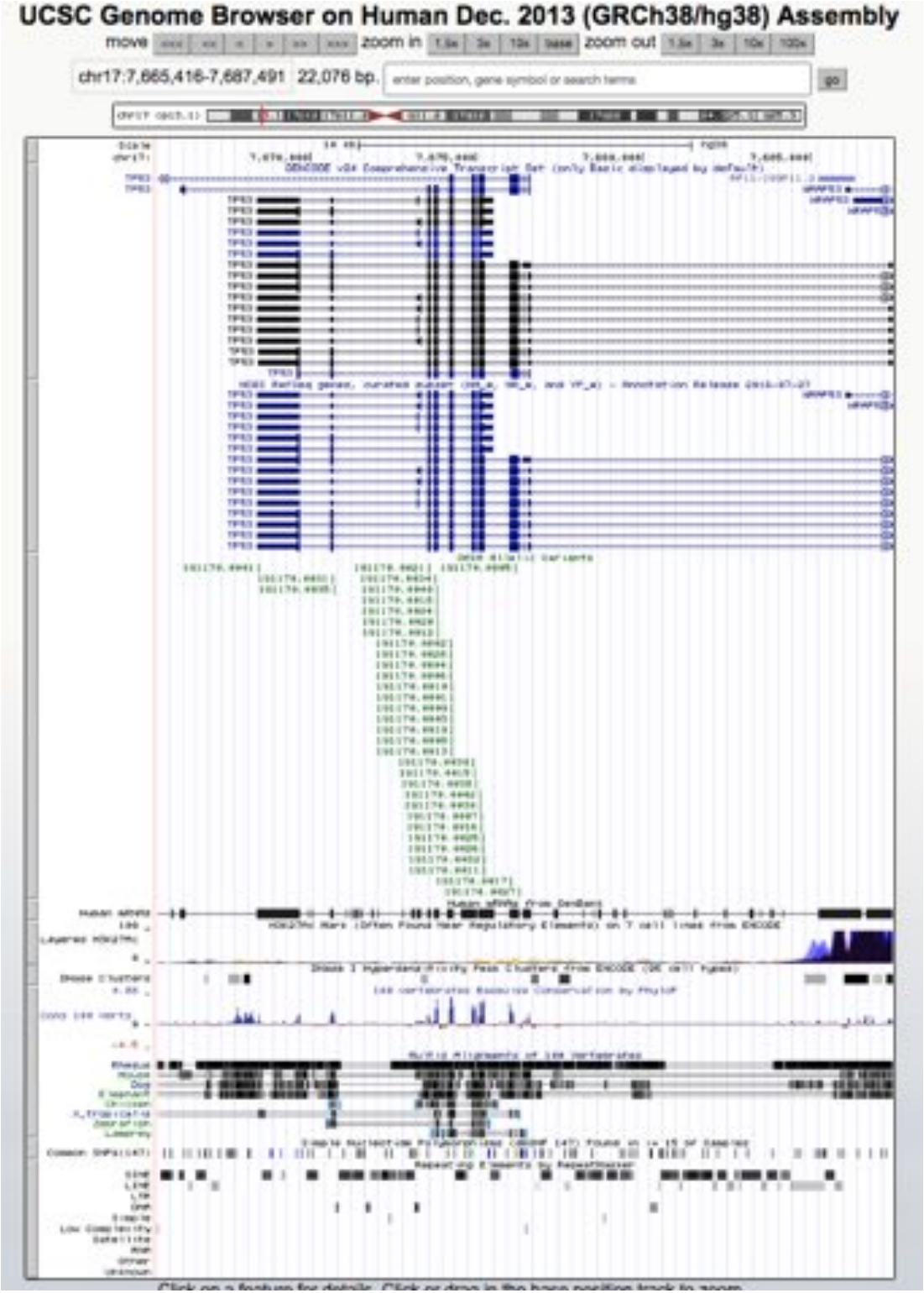


	All INDELs	Valid	PPV	INDELs >5bp	Valid (>5bp)	PPV (>5bp)
Intersection	160	152	95.0%	18	18	100%
WGS	145	122	84.1%	33	25	75.8%
WES	161	91	56.5%	1	1	100%

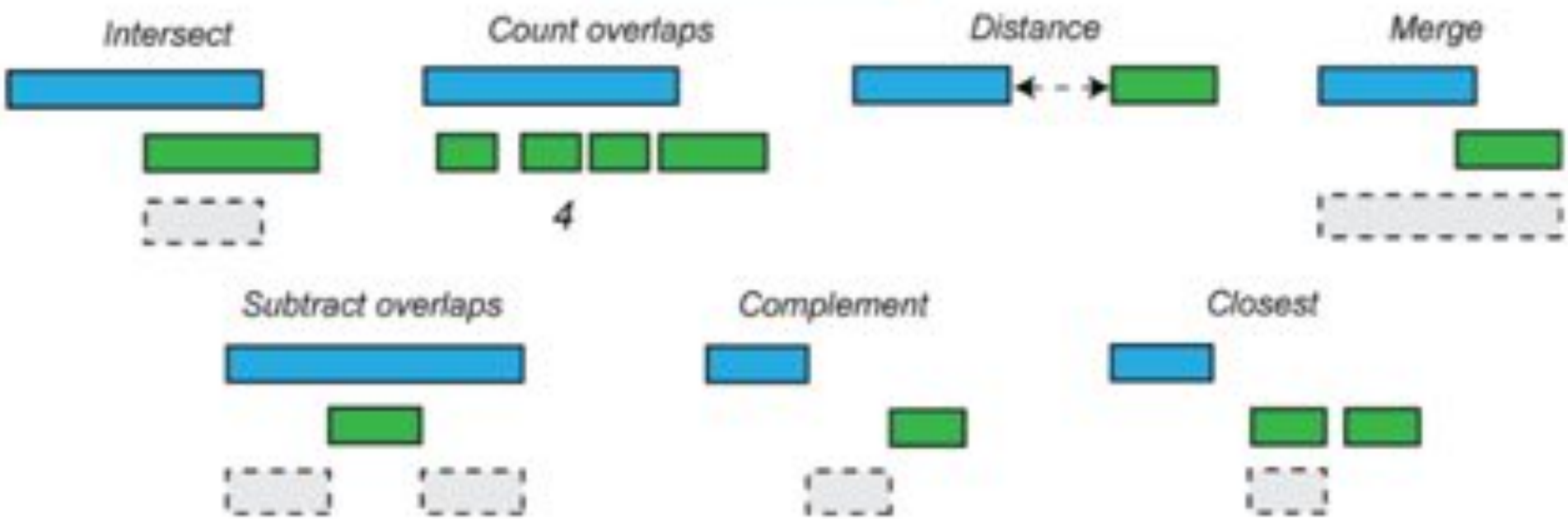
Reducing INDEL calling errors in whole-genome and exome sequencing data
Fang *et al. Genome Medicine* (2014) 6:89. doi:10.1186/s13073-014-0089-z

What are genome intervals?

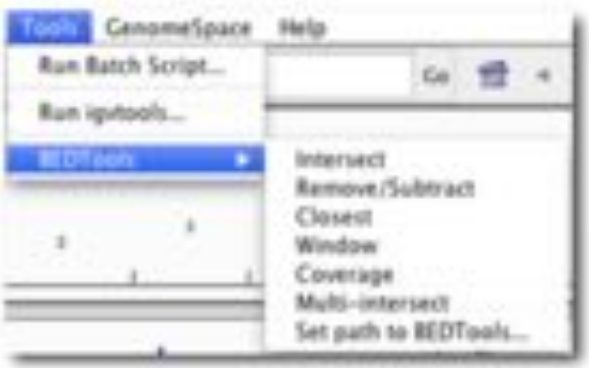
- Genetic variation:
  - SNPs: 1bp
  - Indels: 1-50bp
  - SVs: >50bp
- Genes:
  - exons, introns, UTRs, promoters
- Conservation
- Transposons
- Origins of replication
- TF binding sites
- CpG islands
- Segmental duplications
- Sequence alignments
- Chromatin annotations
- Gene expression data
- …
-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



BEDTools to the rescue!

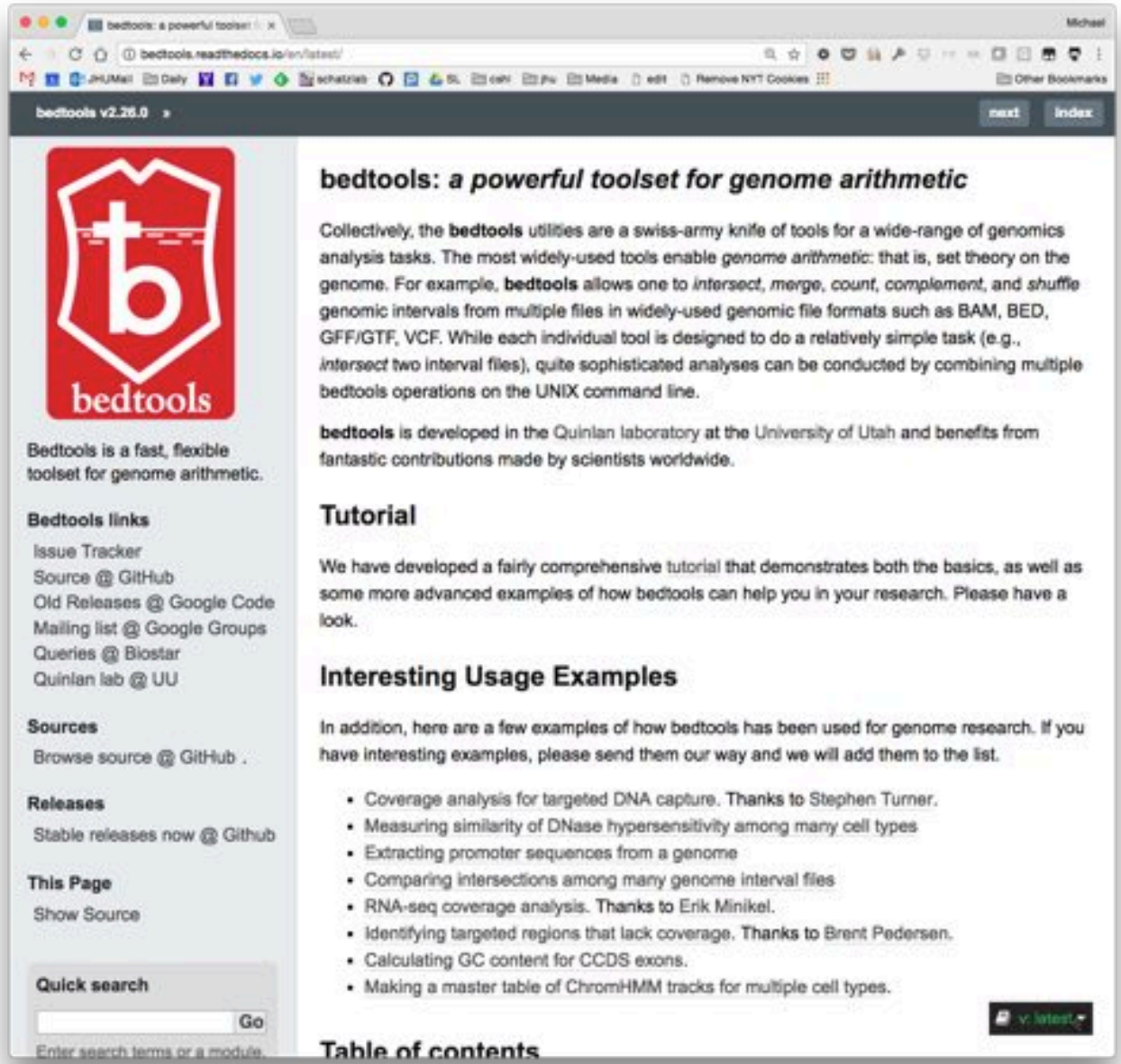


Getting & Using BEDTools



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:

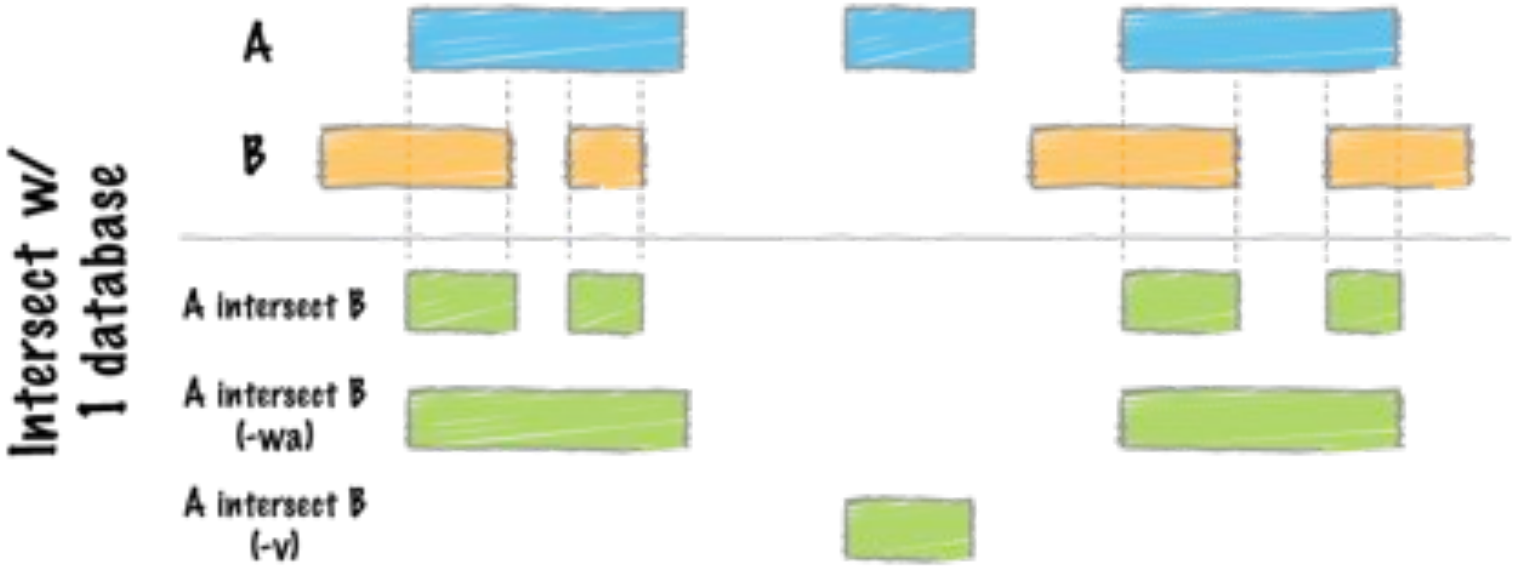
1. chrom — The name of the chromosome (e.g. chr3, chrY, chr2_random) or scaffold (e.g. scaffold10671).
2. chromStart — The starting position of the feature in the chromosome or scaffold. The first base in a sequence is numbered 0.
3. chromEnd — The ending position of the feature in the chromosome or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



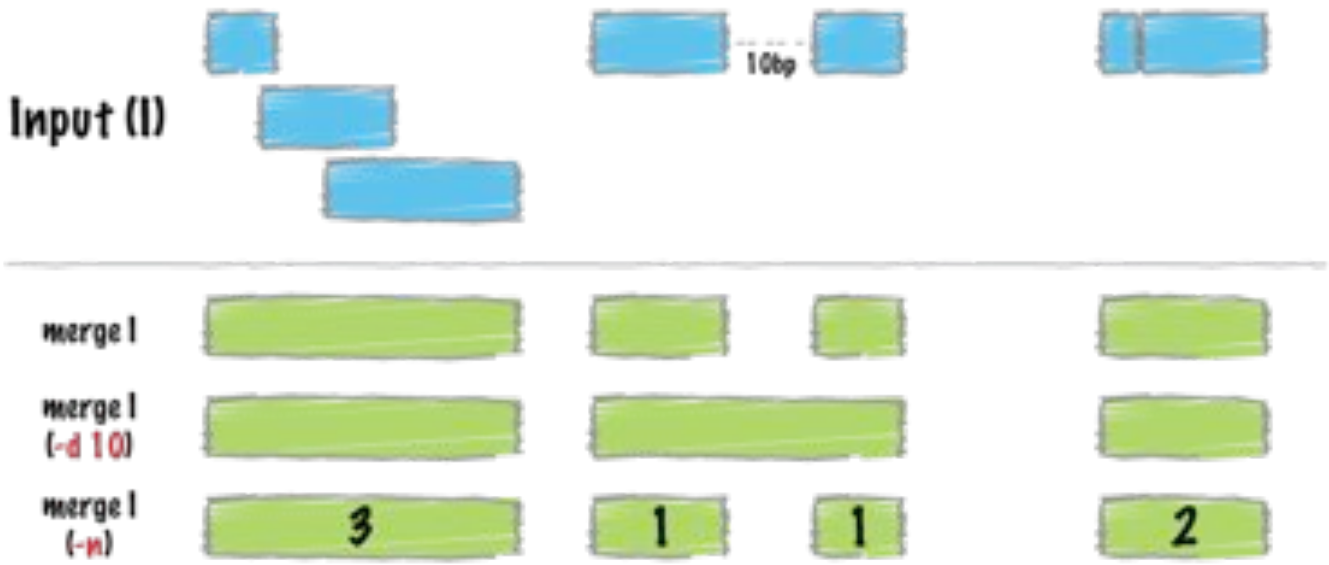
What exons are hit by SVs?

```
$ 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



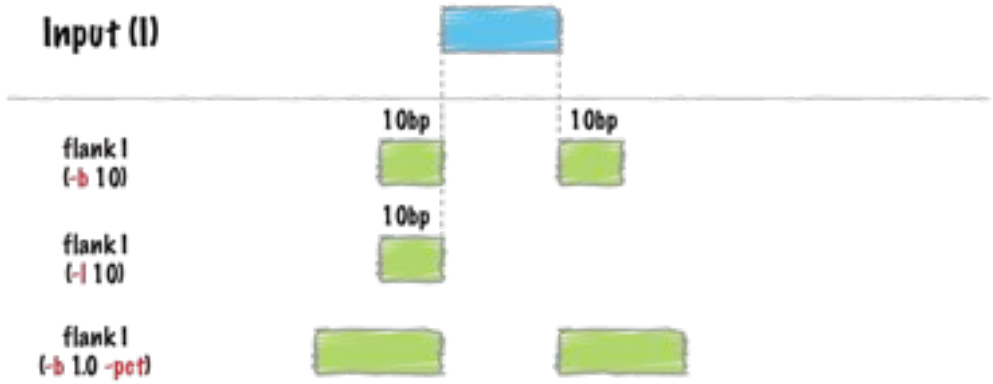
What parts of the genome are exonic?

```
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!

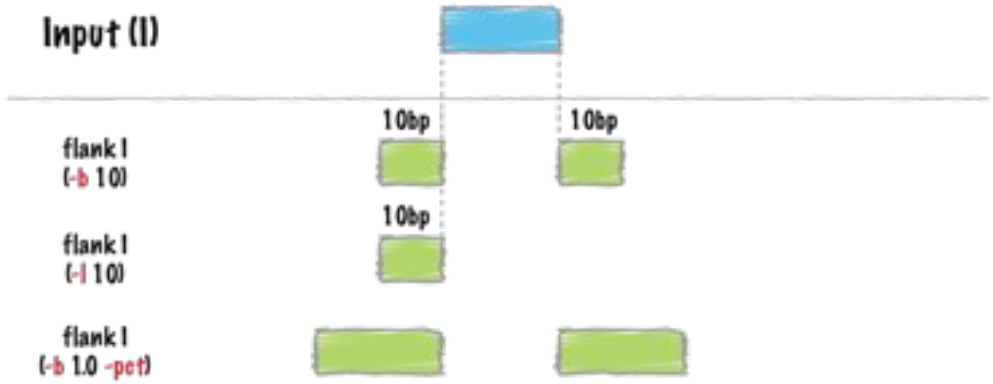
```
sort -k1,1 -k2,2n foo.bed > foo.sort.bed
```

BEDTools 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
```

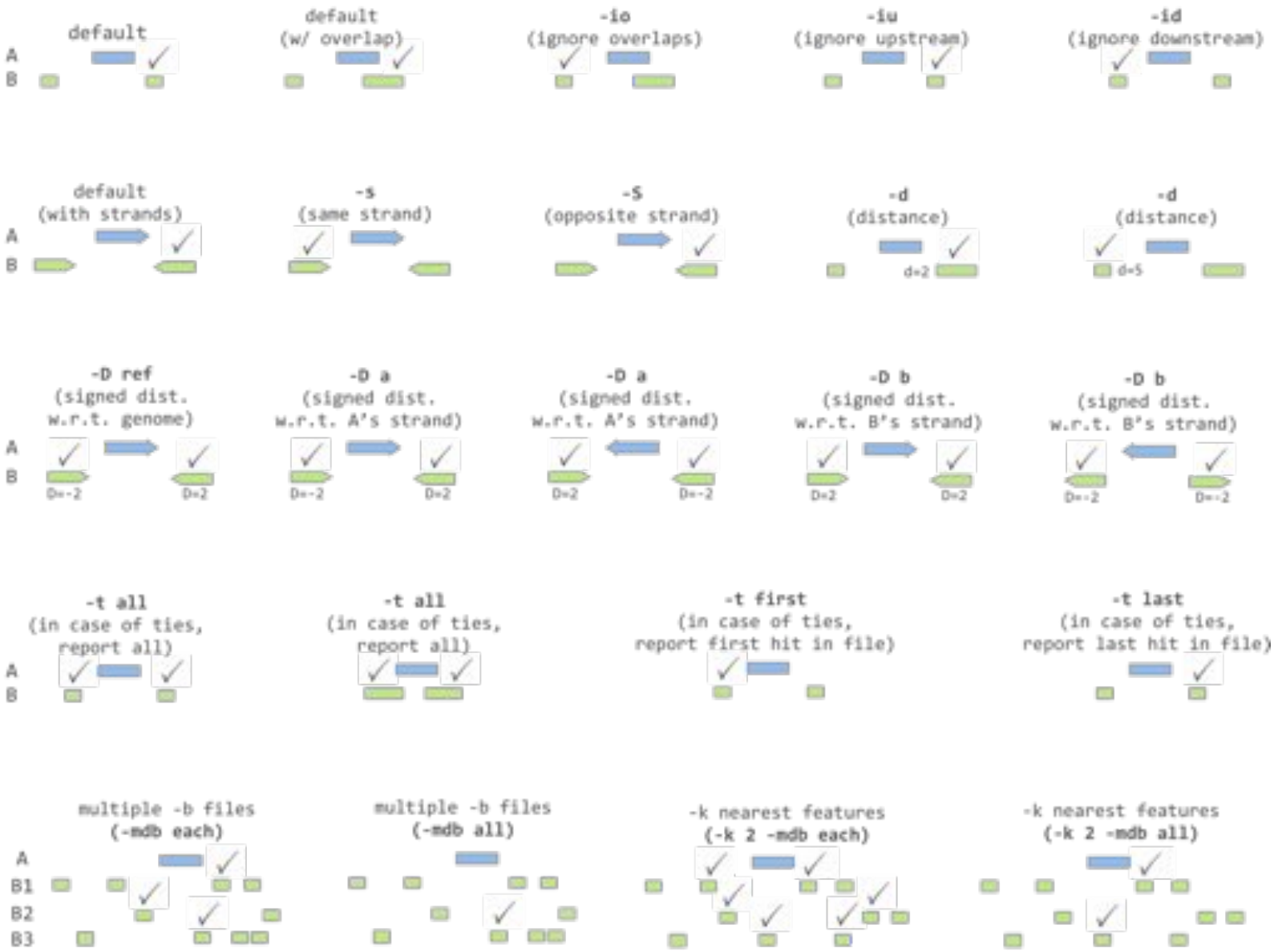
BEDTools Flank & getfasta




```
## 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
```

Can also use the samtools faidx output

BEDTools Closest



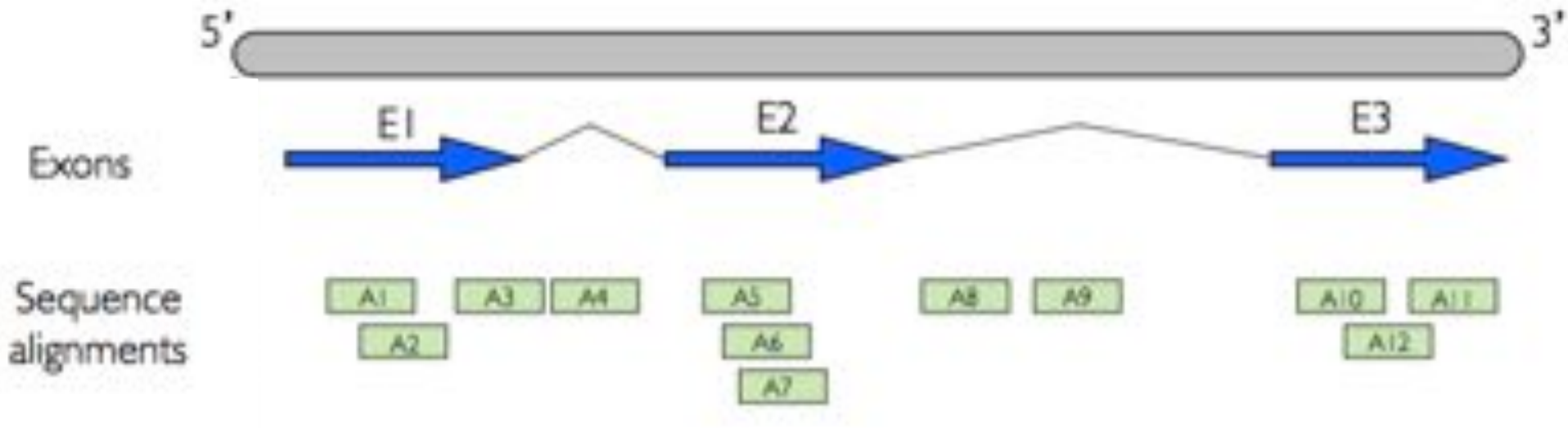
What is the gene closest to this SNP or this enhancer?

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

BEDTools Performance



How many reads are aligned to exonic sequences?

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

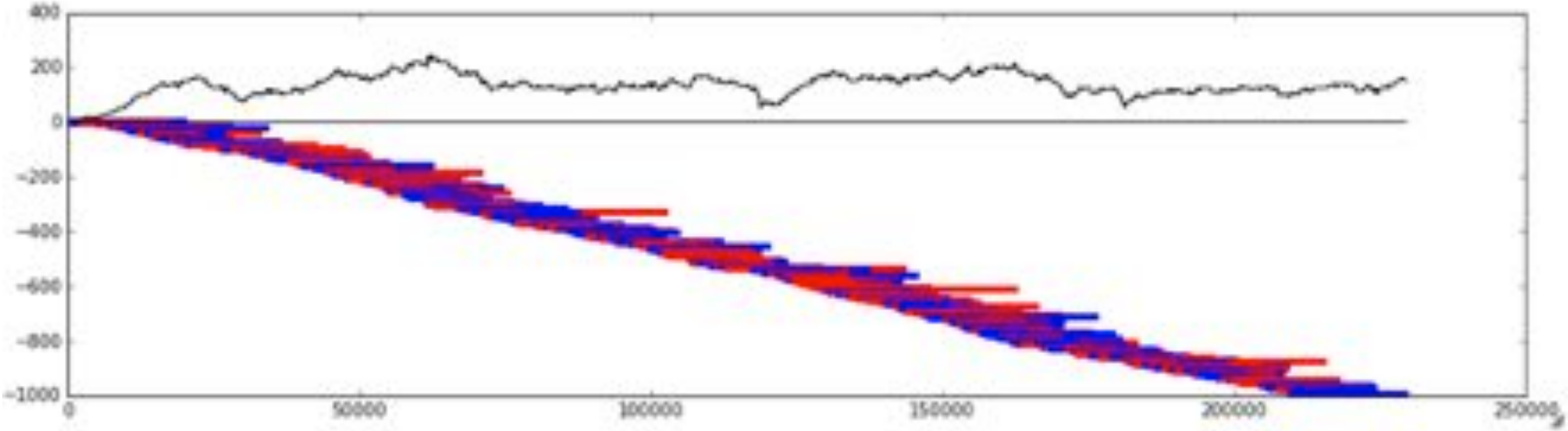
```
if ((read.start <= exon.end) && (read.end >= exon.start)) { print “in exon!”; }
```

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

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

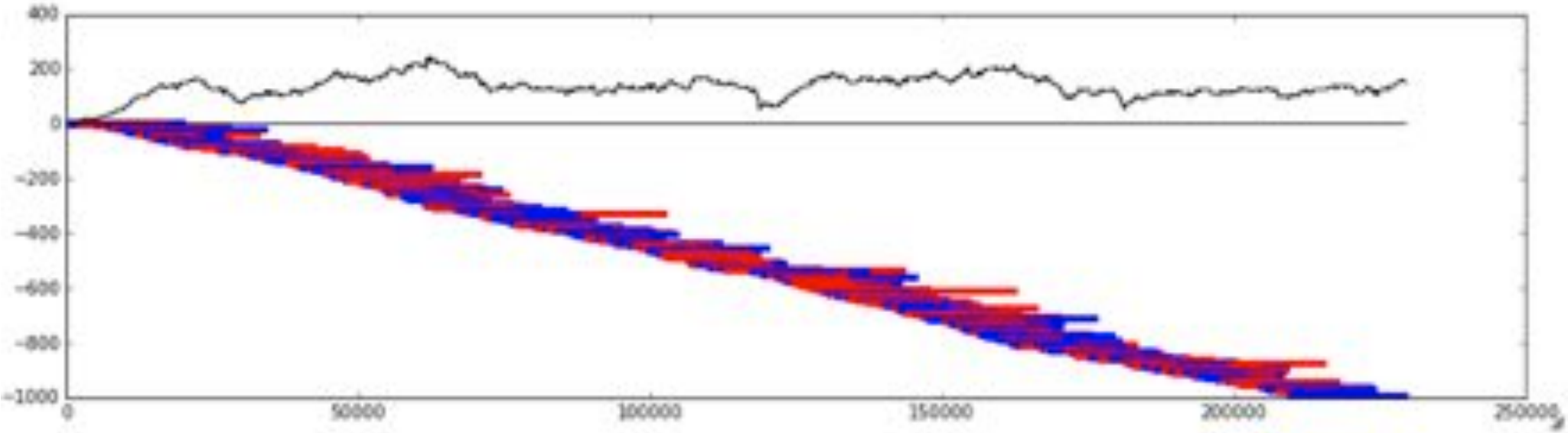
900M reads x 1.1M exons = 990MM comparisons! ☹

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]
```

* This is what you should do for the homework! *

Notice that it took 4435 ms for this to complete

Delta Encoding
aka run length encoding

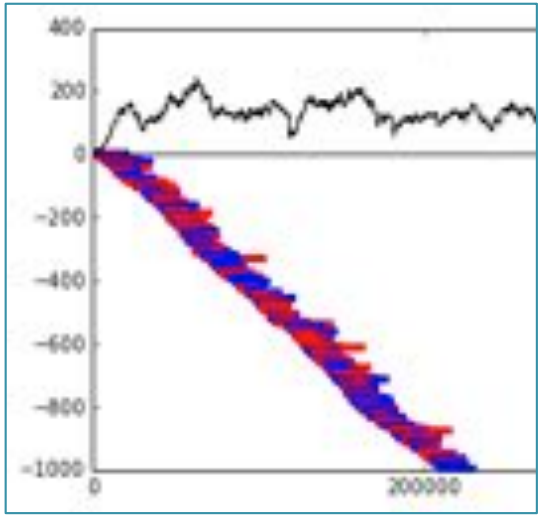
```
deltacov = []
curcov = -1
for i in xrange(0, len(brutecov)):
  if brutecov[i] != curcov:
    curcov = brutecov[i]
    delta = (i, curcov)
    deltacov.append(delta)

## 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")
```

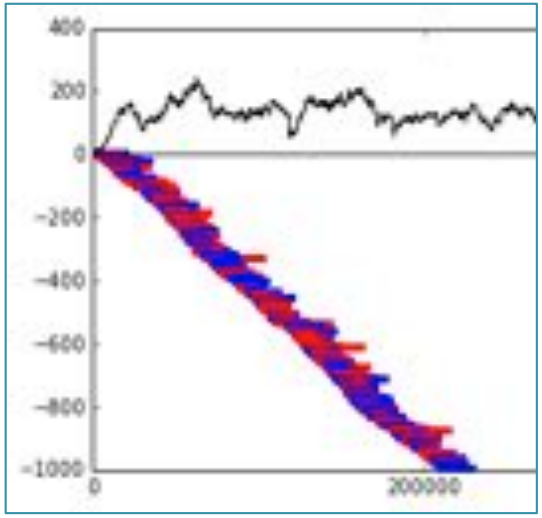


Plot Each Read

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

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

The basic algorithm works like this:

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

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

- walking from start position to start position
  - check to see if we past any read ends
  - coverage goes down by one when a read ends
  - coverage goes up by one when new read is encountered

Plane Sweep

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

       1
```

arrive at r1 [1,30]:

active set is empty; add to active set: 30

output (1,1)

Plane Sweep

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

       1    2
```

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 at beginning of active set

output (5, 2)

Plane Sweep

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

       1    2    3
```

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 25 to active set: 25, 30 <- notice insert at beginning of active set

output (5, 2)

arrive at r3 [10,45]:

10 < 25; add 45 to active set: 25, 30, 45 <- add to end of active set

output (10, 3)

Plane Sweep

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

       1    2    3         4
```

arrive at r3 [10,45]:

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

arrive at r4 [20,40]:

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

Plane Sweep

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

       1    2    3         4    3
```

arrive at r5[35,55]:

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

output (25, 3)

Plane Sweep

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

       1    2     3           4     3     2
```

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

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

       1    2     3           4     3     2     3
```

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

Plane Sweep

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

       1    2     3           4     3     2     3
```

Flush: 40, 45, 55

Plane Sweep

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

       1    2     3           4     3     2     3     2
```

Flush: 40, 45, 55

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

Plane Sweep

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

       1    2     3           4     3     2     3     2     1
```

Flush: 40, 45, 55

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

step down at 45: active set: 55
output (45, 1)

Plane Sweep

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

       1    2     3           4     3     2     3     2     1          0
```

Flush: 40, 45, 55

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

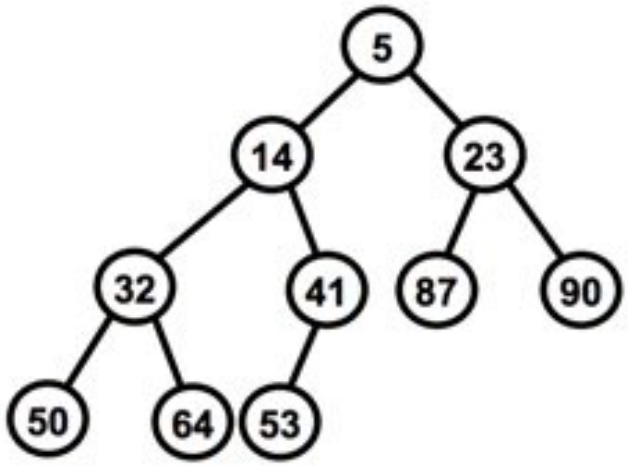
Do we really need the whole list to be sorted?

```
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



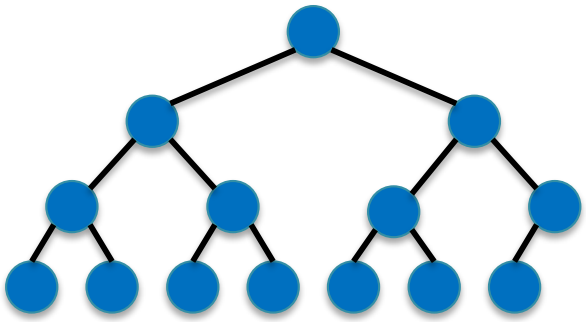
Very efficient data structure for dynamically maintaining a set of element while allowing you to find the minimum (or maximum) very fast:

Insert: O(lg(n)) <- super fast
Remove: O(lg(n)) <- super fast
Find-min: O(1) <- instantaneous

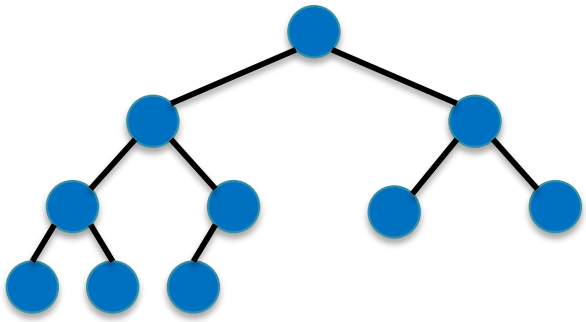
Key to fast performance derives from ***heap shape property***: the tree is guaranteed to be a complete binary tree, meaning it will remain balanced and the height will always be log(n)

Binary Heaps

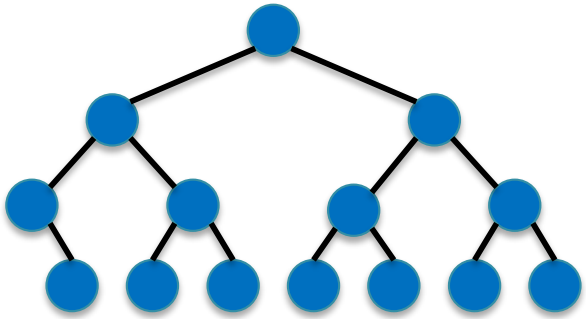
Shape Property:
Complete binary tree with every level full, except potentially the bottom level, **AND** bottom level filled from left to right



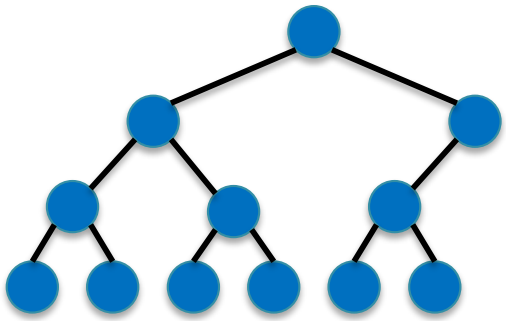
Valid



Valid



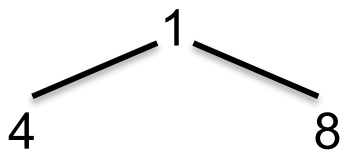
Invalid



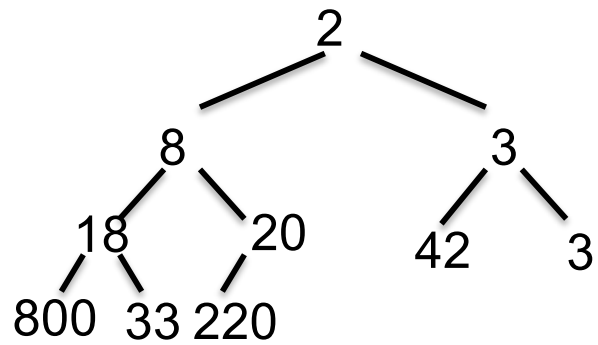
Invalid

Min Binary Heaps

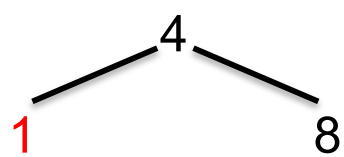
Ordering Property:
The value of each node is less than or equal to the value of its children,
BUT there is no ordering between left and right children



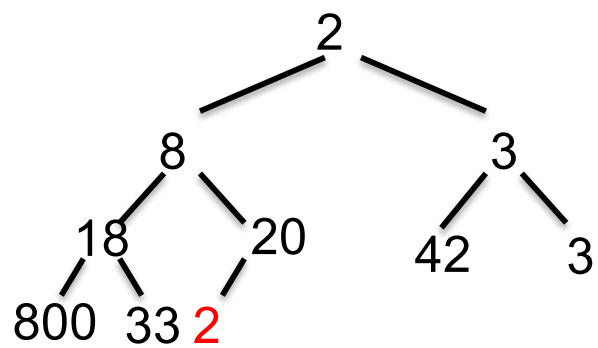
Valid



Valid

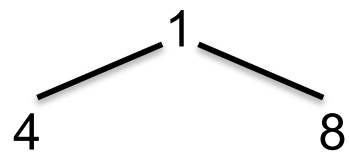


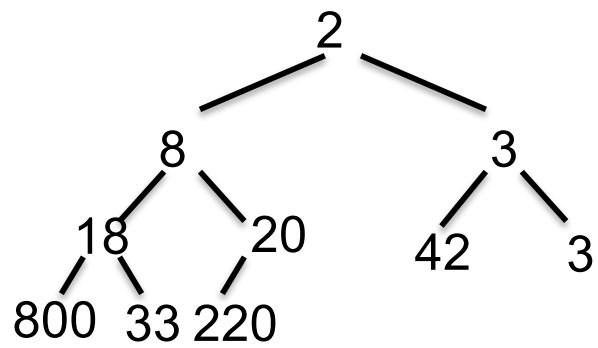
Invalid



Invalid

Min Binary Heaps





What does the <u>shape</u> property imply about the <u>height</u> of the tree?

Guaranteed to be lg n ☺

What does the <u>ordering</u> property imply about the root of the tree?

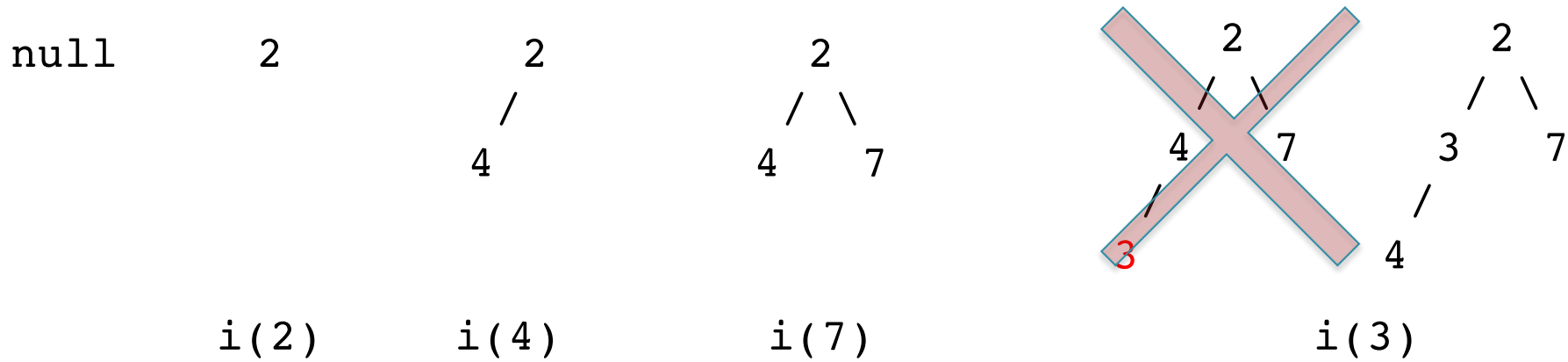
Guaranteed min (or max) value will be in the root node

That's interesting, I wonder if we could use this for a priority queue…

… just need to efficiently insert() and removeTop()

Inserting into a binary heap

Insert the elements 2, 4, 7, 3



The ***shape property*** tells us that we need to fill one level at a time, from left to right. So the ***number of elements*** in a heap ***uniquely determines where the next node*** has to be placed.

What about the ***ordering property***? When we insert 3, the parent 4 so the ***ordering property is violated***. There's an ***easy fix*** however, just swap the values!

Note that in general, we ***may need to keep swapping "up the tree"*** as long as the ordering property is still violated. ***But since there are only log n levels, this can take at most O(log n) time in the worst case.***

Remove top from a binary heap

Remove the top

```
      2
    /   \
   7     4
  /
 10
```

```
   7     4
  /
 10
```

ERROR:
2 trees

```
      7
    /   \
   10    4
```

ERROR:
4 < 7

```
      7
     /
    10
   /
  4
```

ERROR:
Shape Violation

Any ideas?

1.Swap last

```
      10
    /   \
   7     4
  /
 2
```

2. Remove last

```
      10
    /   \
   7     4
```

3. Swap down from root with smaller child

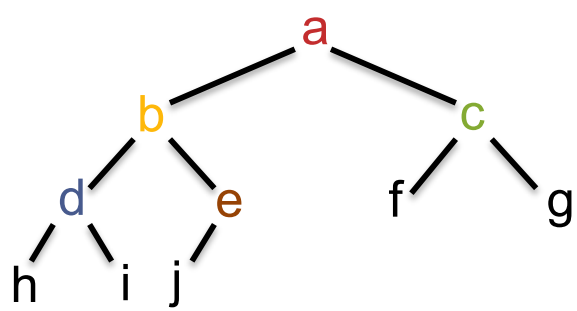
```
      4
    /   \
   7     10
```

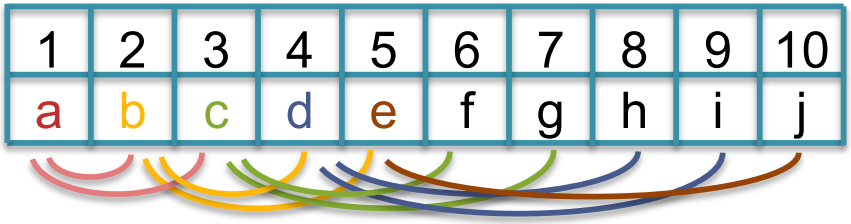
Note that in general, we ***may need to keep swapping "down the tree"*** as long as the ordering property is still violated. ***But since there are only log n levels, this can take at most O(log n) time in the worst case.***

Heap Implementation

We could implement a heap as a tree with references, but those references take up a lot of space and are relatively slow to resolve

Lets encode the tree inside an array!





Encoding a complete tree into the array in <u>level order</u> puts the children and parent in predictable locations (Math is easier if the array starts at 1 instead of 0)

Parent(i) = array[i/2]
Parent(f) = parent(6) = array[6/2] = array[3] = c

left(i) = array[i*2] & right(i) = array[i*2+1]
left(3) = array[3*2] = array[6] = f & right(3) = array[3*2+1] = array[7] = g

Heaps In Python

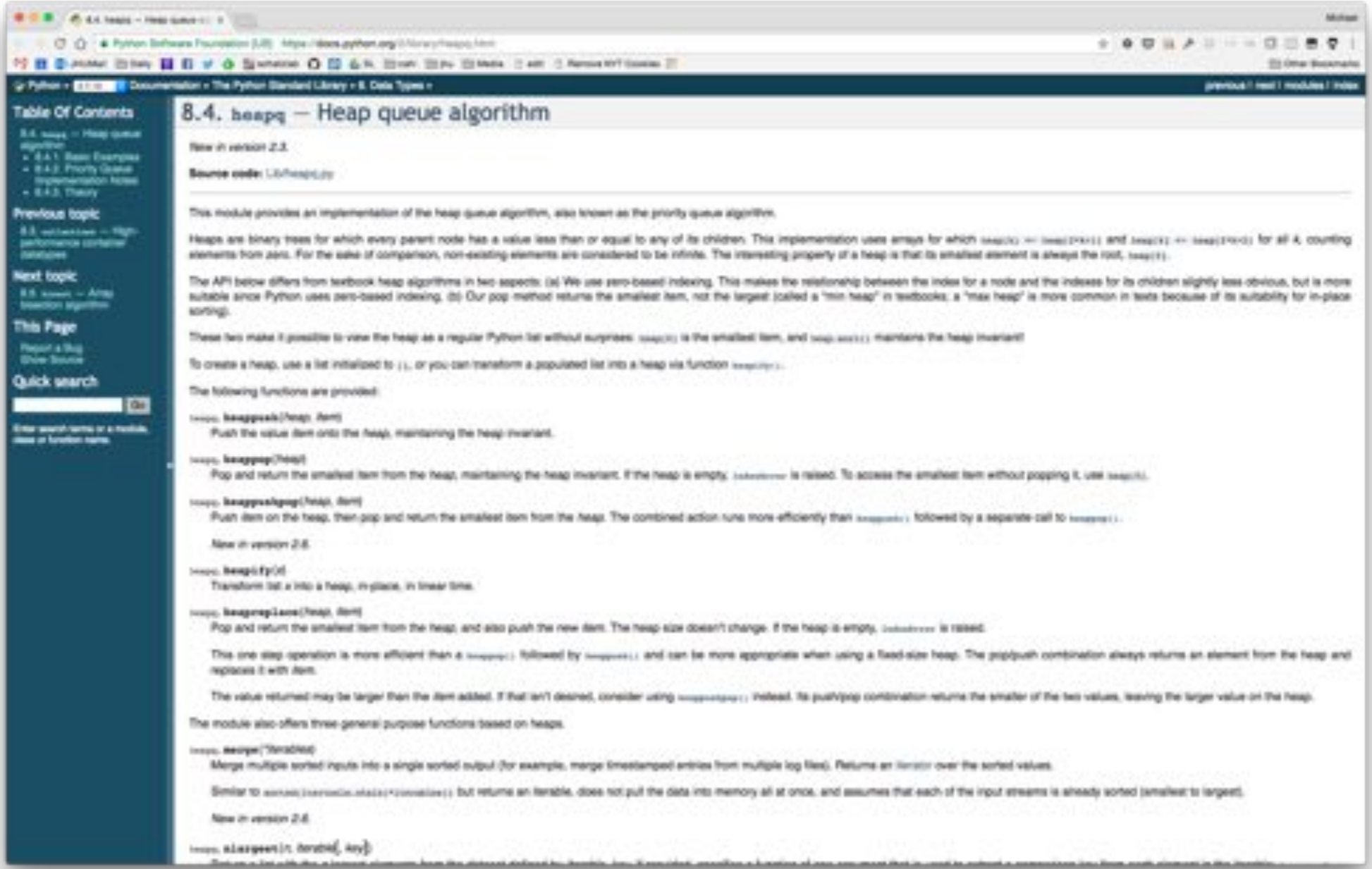


Table Of Contents

Previous topic

Next topic

This Page

Quick search

8.4. heapq — Heap queue algorithm

New in version 2.3.

Source code: Lib/heapq.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.**heapreplace**(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])

Heap-based Plane-Sweep

```python
## 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

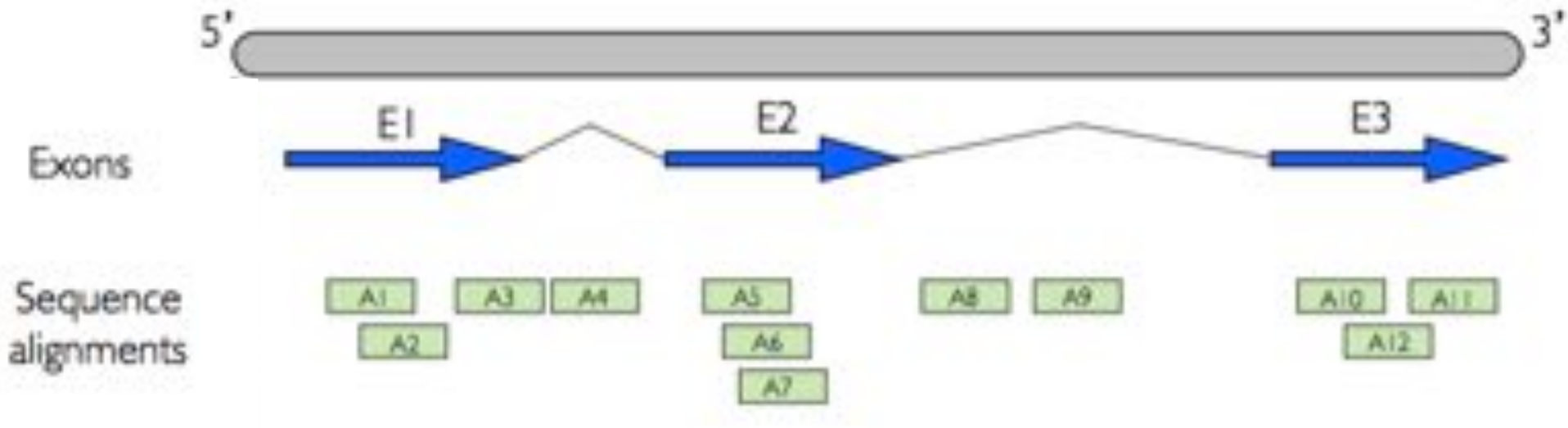
planeheap[0] is min

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)!
```

BEDTools Performance



How many reads are aligned to exonic sequences?

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

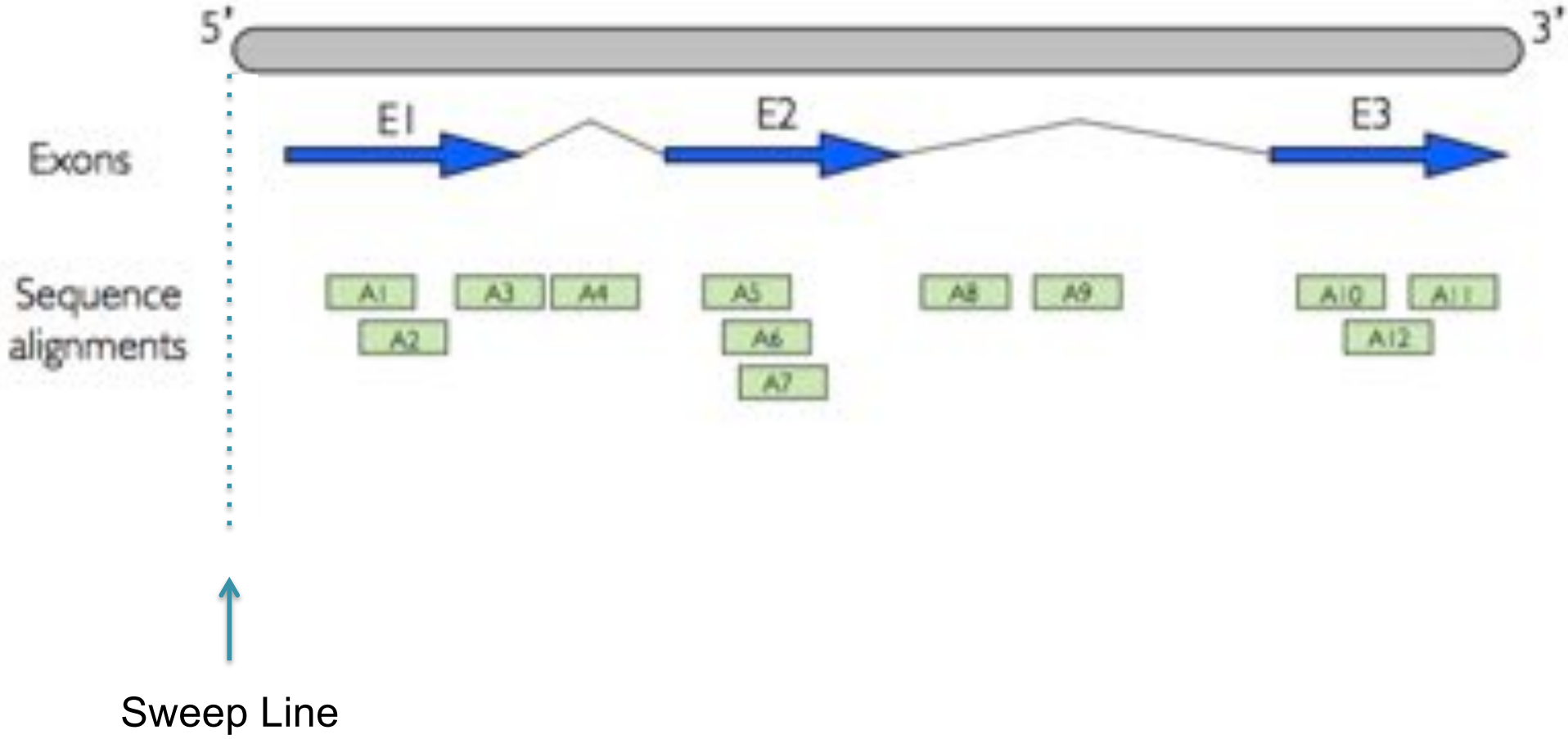
```
if ((read.start <= exon.end) && (read.end >= exon.start)) { print “in exon!”; }
```

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

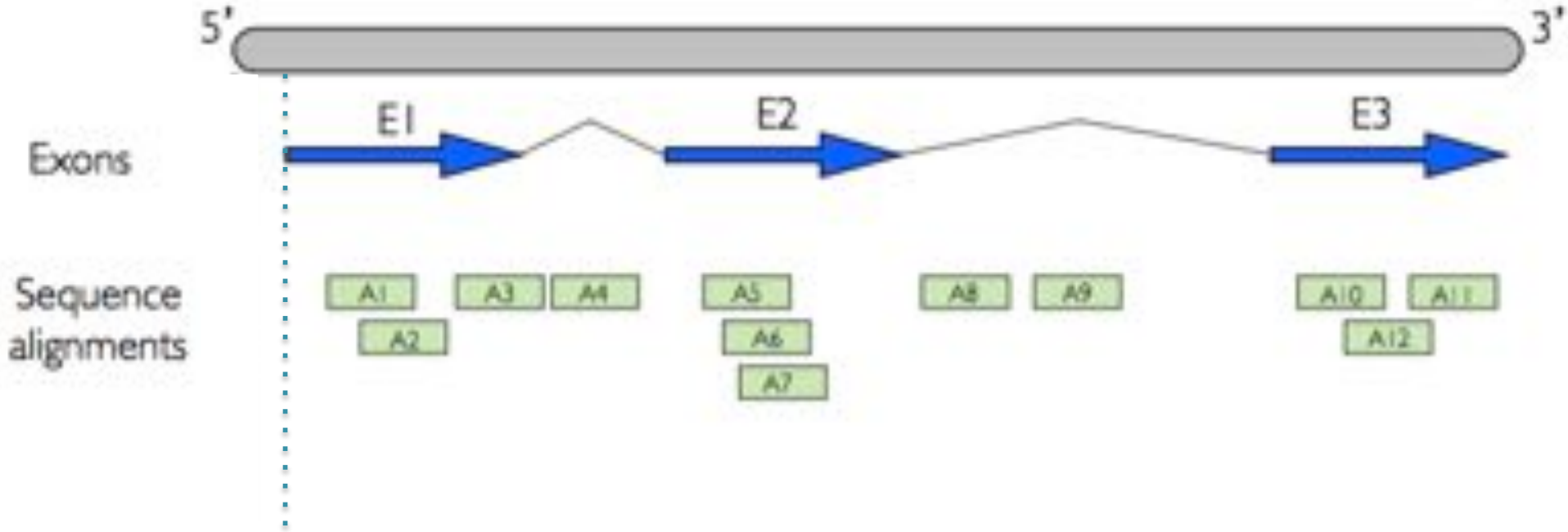
30x3Gb = 90Gbp / 100bp reads = 900M reads

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

Plane Sweep to the Rescue!



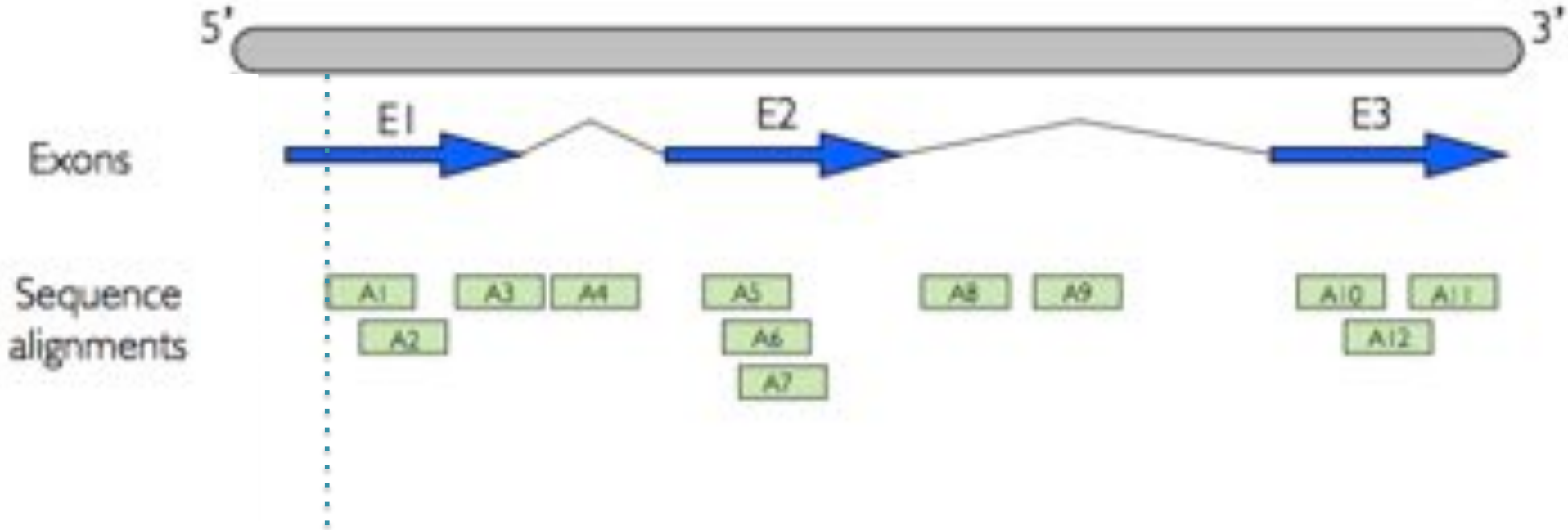
Plane Sweep to the Rescue!



<u>Start</u> of E1
E1 is active

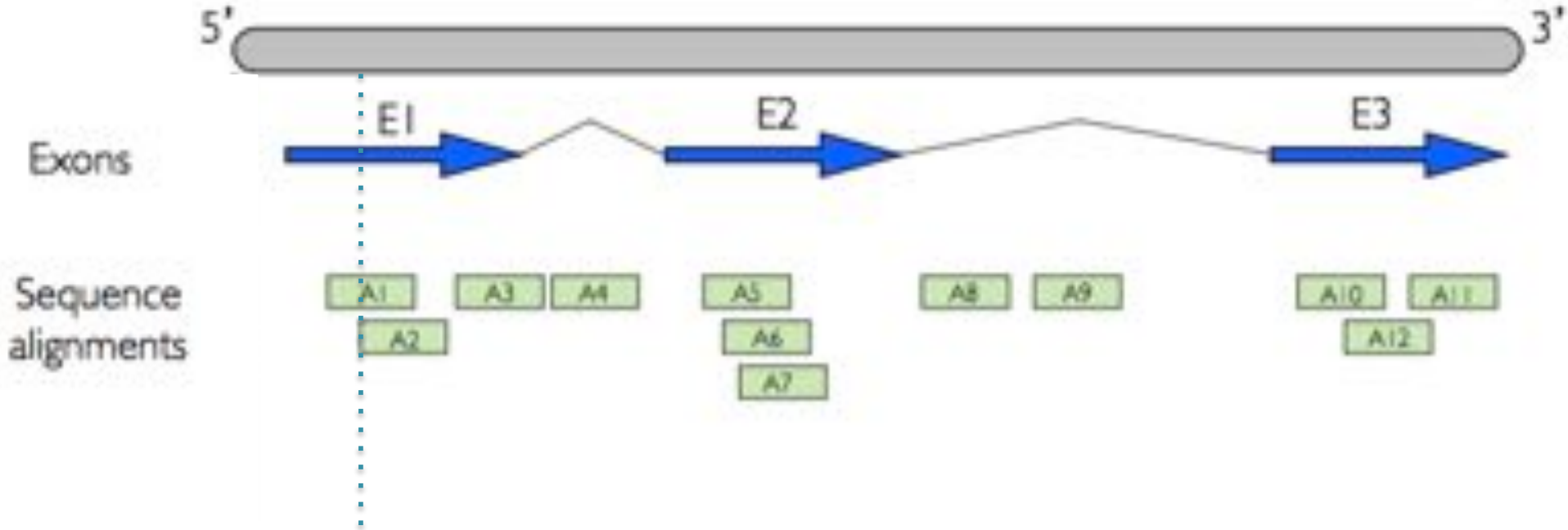
{E1}

Plane Sweep to the Rescue!




```
A1 overlaps E1

  {E1=(A1)}
```

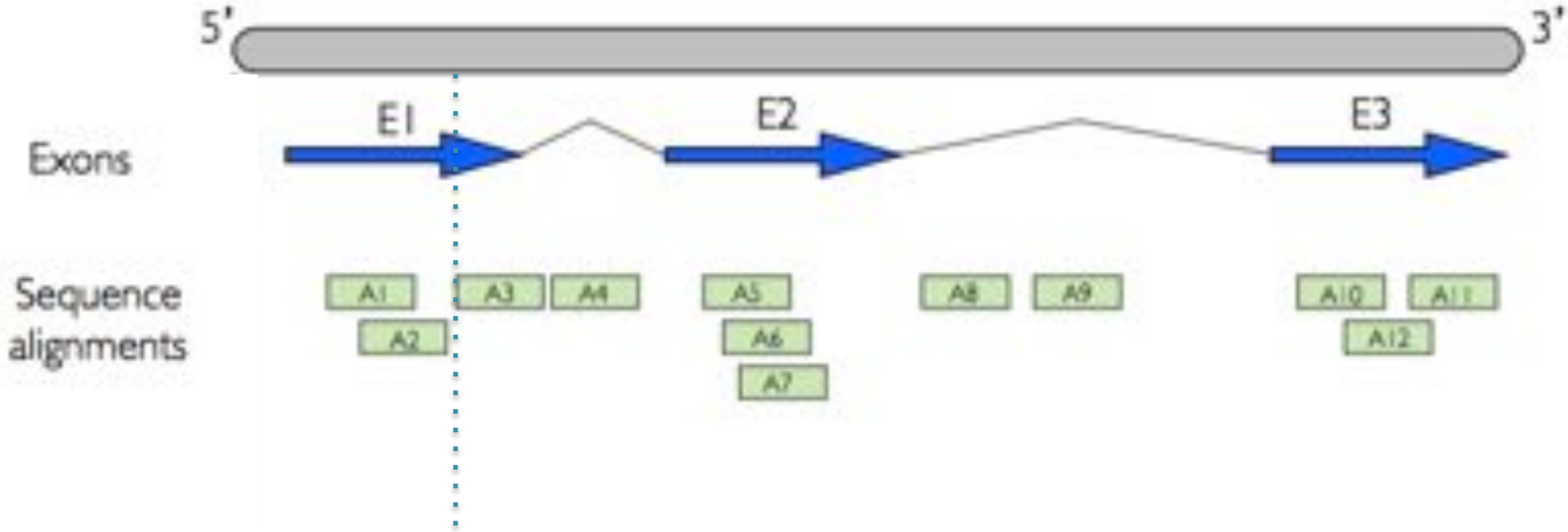
Plane Sweep to the Rescue!



A2 overlaps E1

{E1=(A1, A2)}

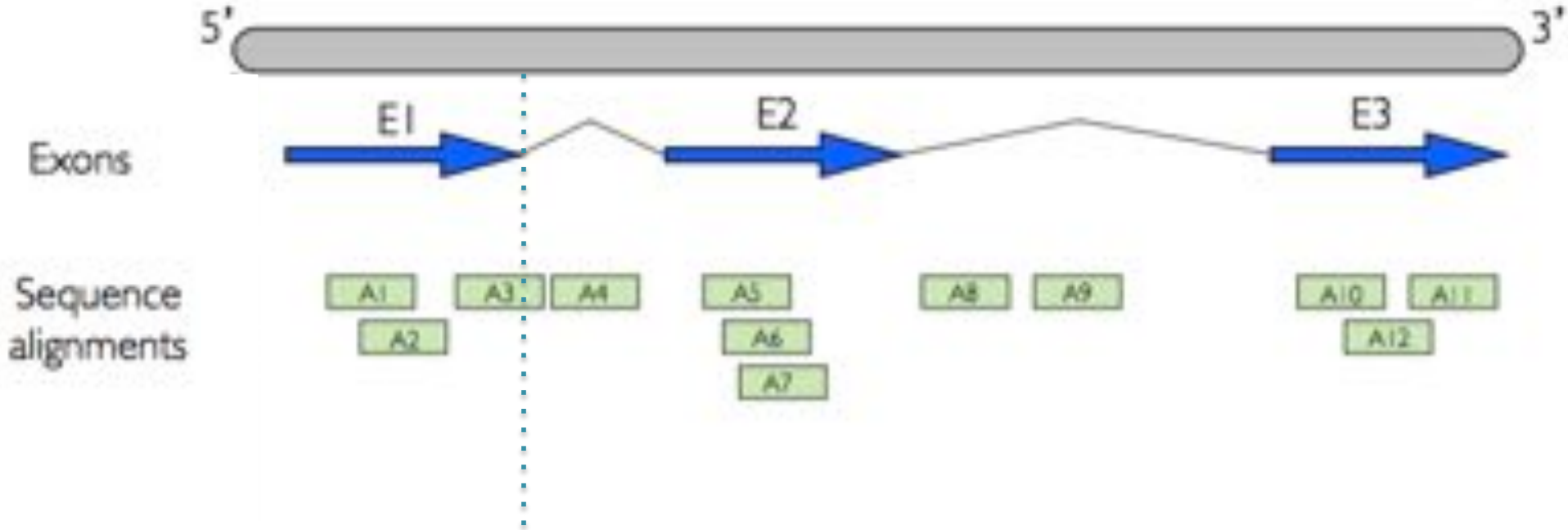
Plane Sweep to the Rescue!



A3 overlaps E1

{E1=(A1, A2, A3)}

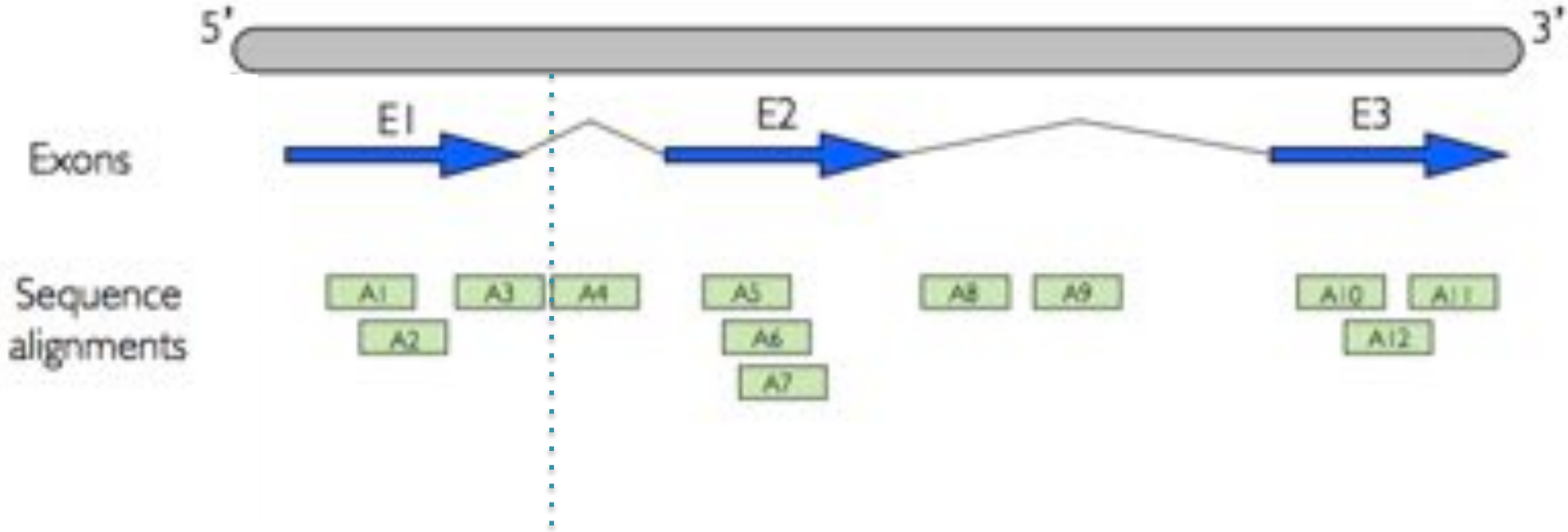
Plane Sweep to the Rescue!



<u>End</u> of E1

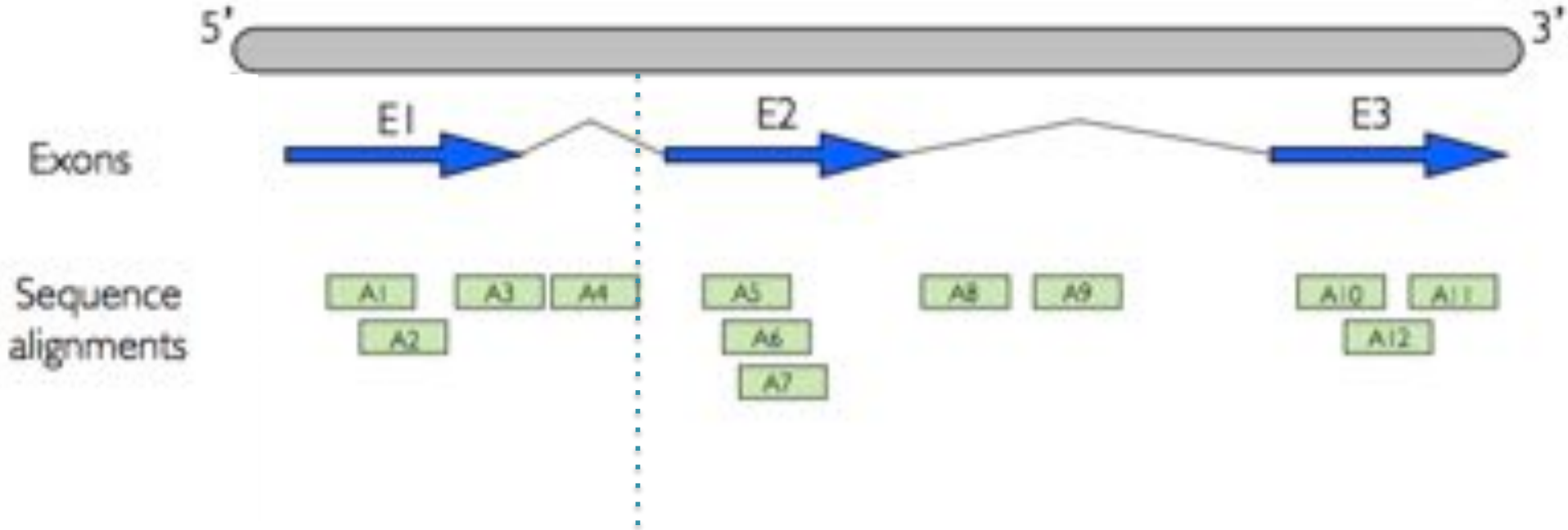
Report:
{E1=(A1, A2, A3)}

Plane Sweep to the Rescue!



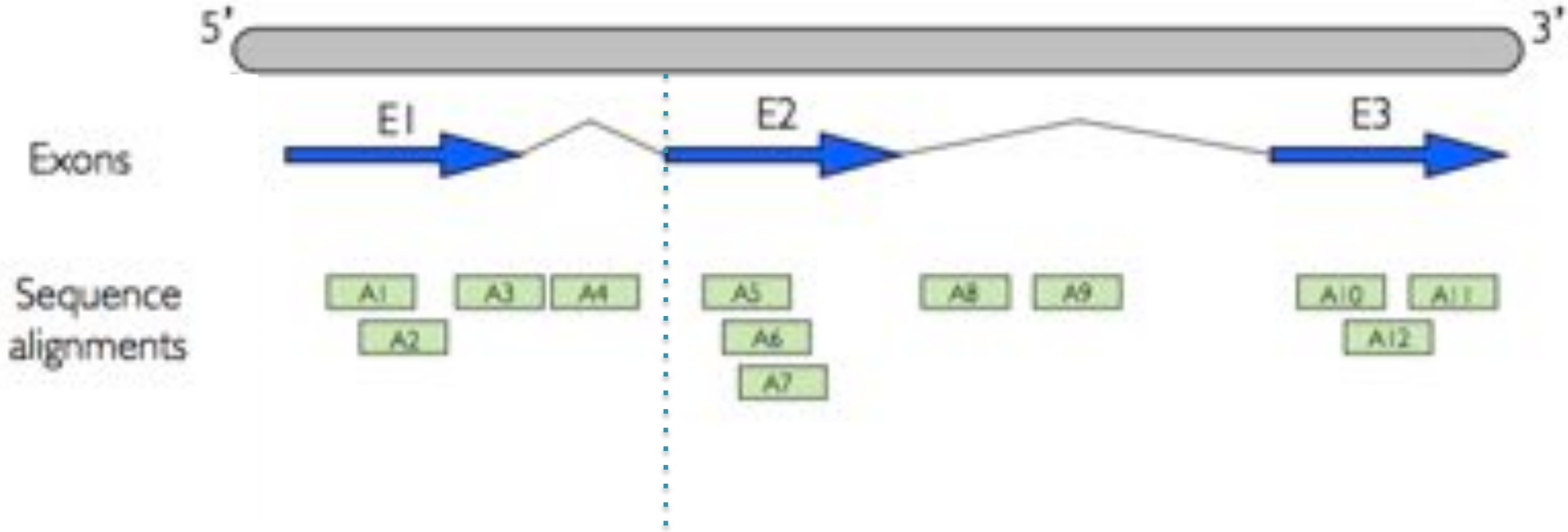
A4 starts,
but nothing is active

Plane Sweep to the Rescue!



A4 end,
but nothing is active

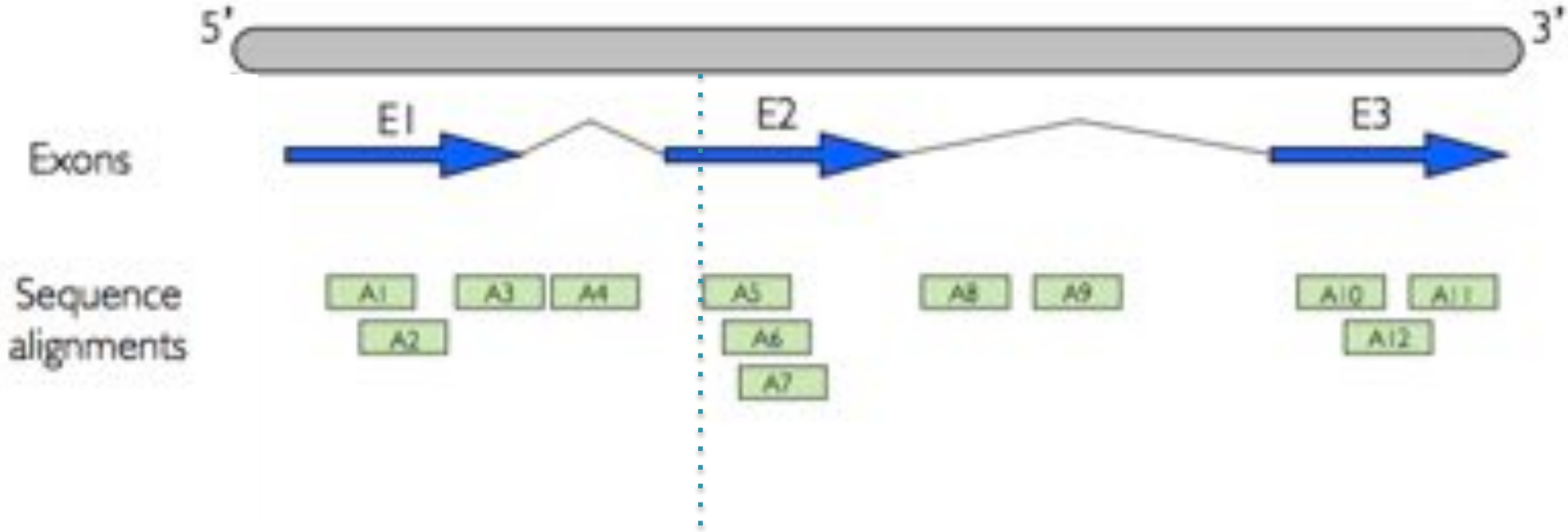
Plane Sweep to the Rescue!



E2 starts

{E2}

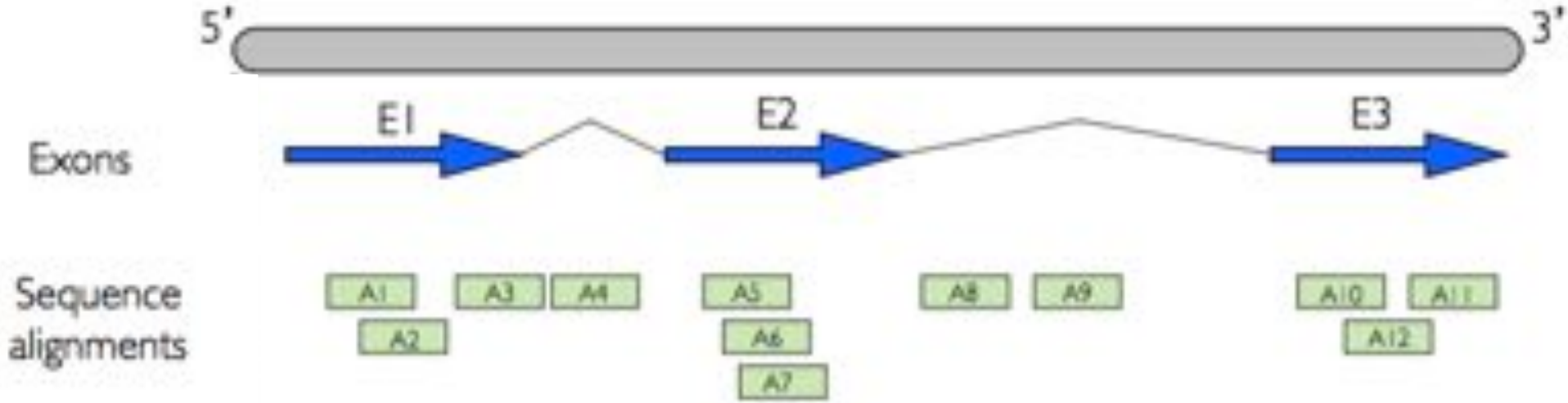
Plane Sweep to the Rescue!



A5 overlaps E2

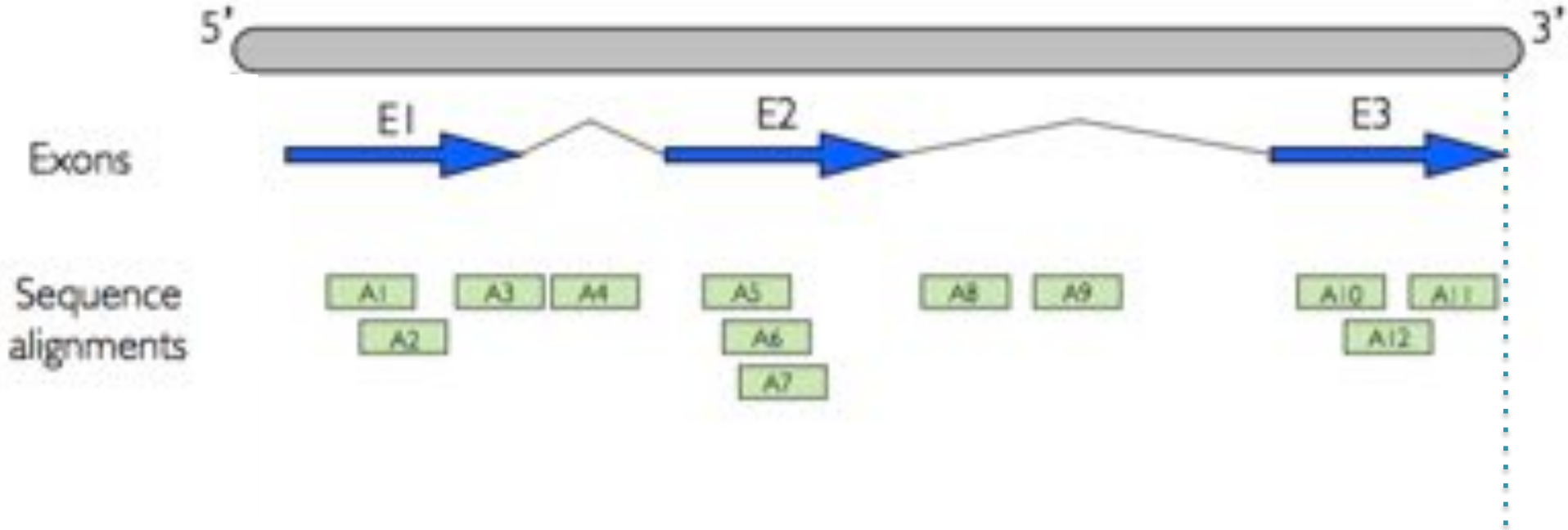
{E2=(A5)}

Plane Sweep to the Rescue!



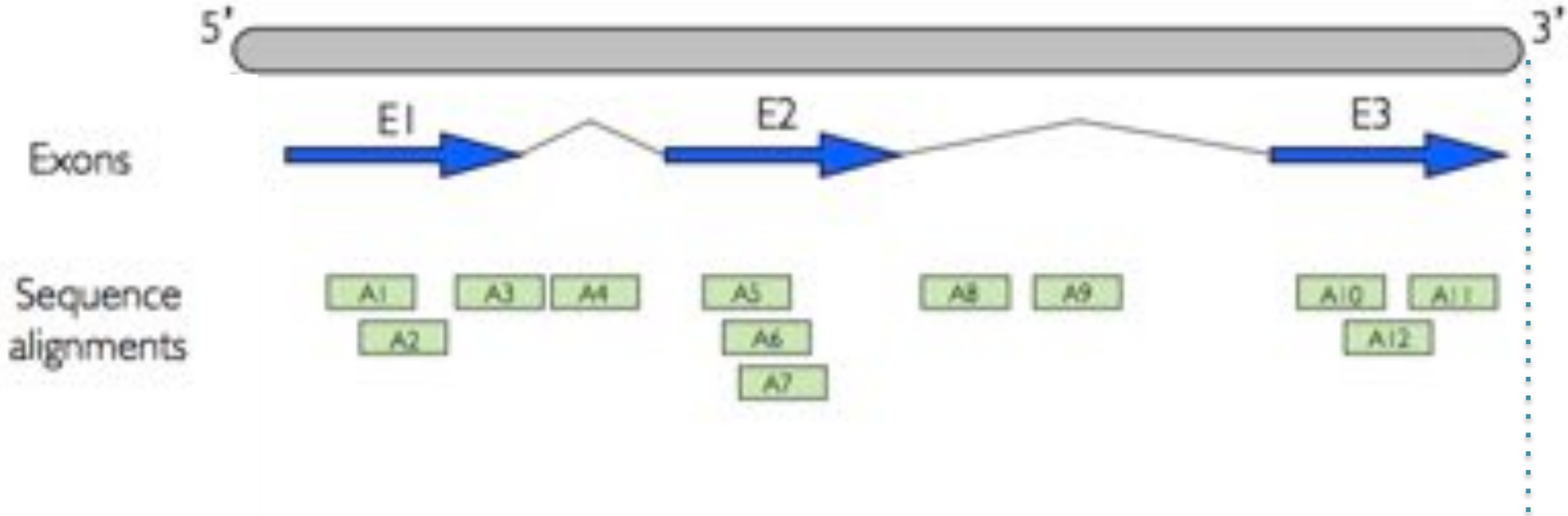
...

Plane Sweep to the Rescue!



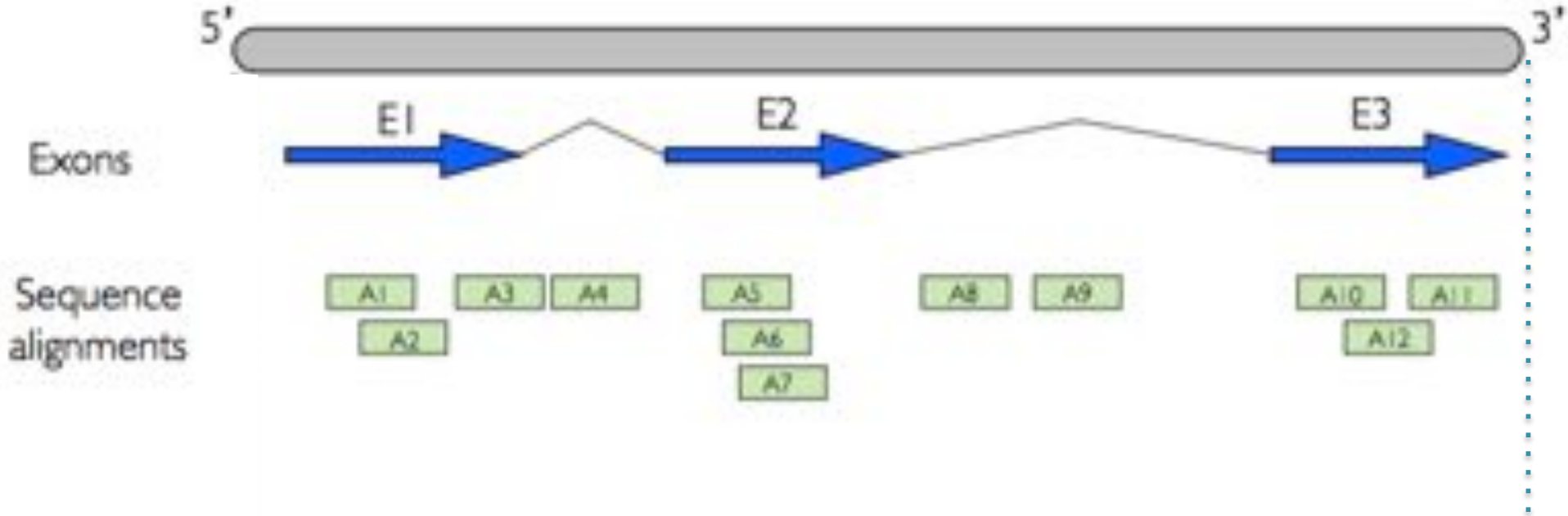
E3 Ends

Plane Sweep to the Rescue!



All done!

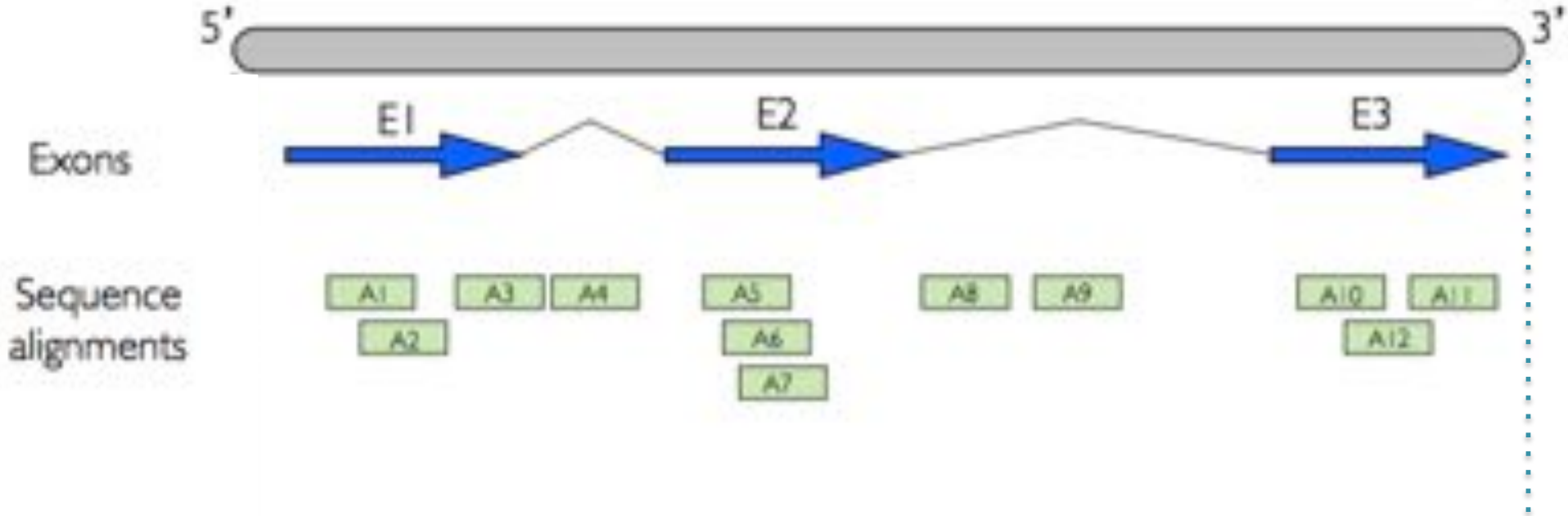
Plane Sweep to the Rescue!



Final Results:

E1=(A1,A2,A3)
E2=(A5,A6,A7)
E3=(A10,A12,A11)

Plane Sweep to the Rescue!



How many comparisons does the plane sweep algorithm make?

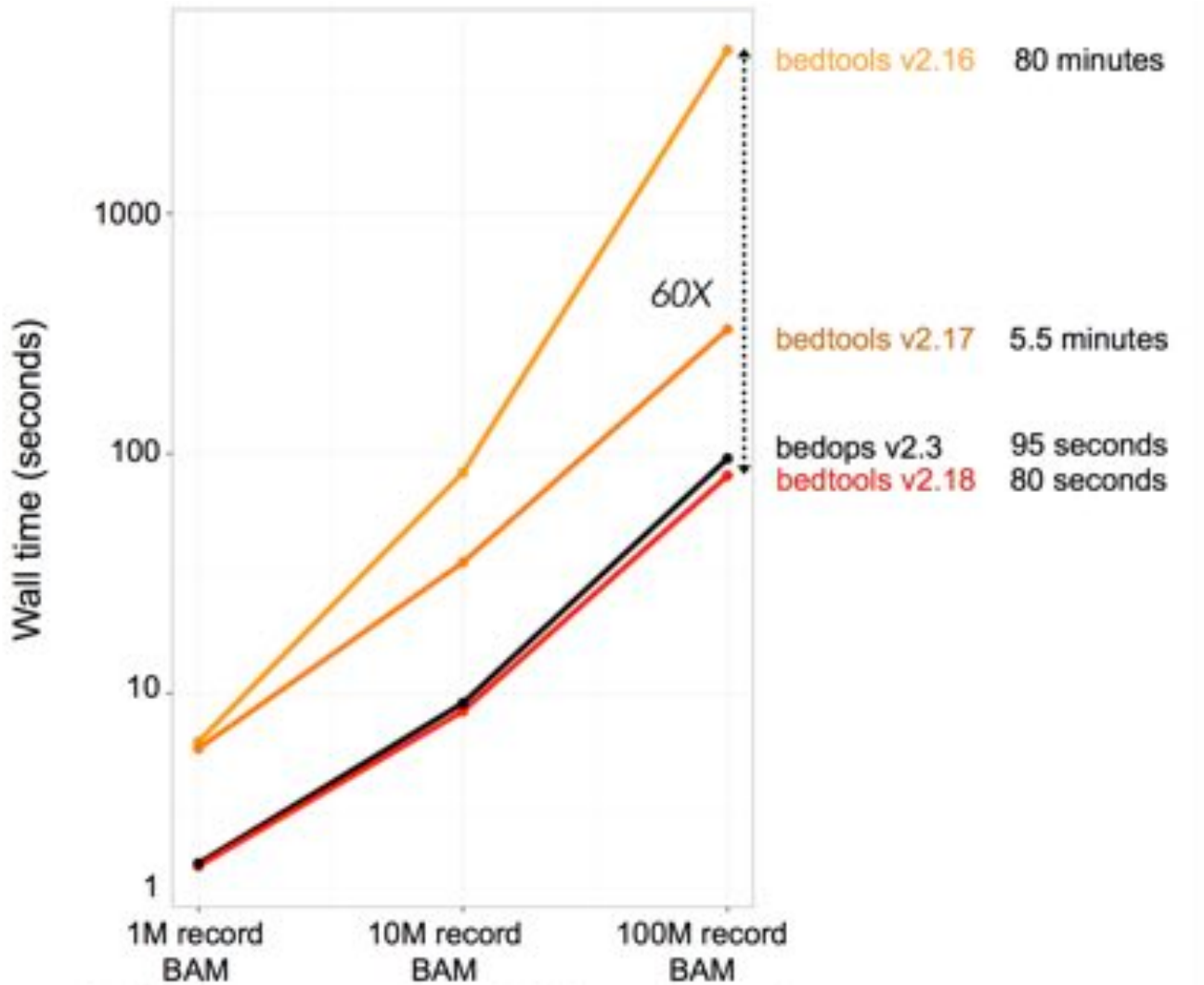
Each read is compared to the “active set”

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

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

Output is basically as fast as we can read the input data ☺

BEDTools Performance




```
bedtools intersect -a gencode.exons.bed -b exome.bam -sorted -c
bedmap --echo --count gencode.exons.bed exome.bam
```

Next Steps

1. See Lecture Notes for Full Details
2. Review Bedtools docs: http://bedtools.readthedocs.io/
3. Finish Assignment 3