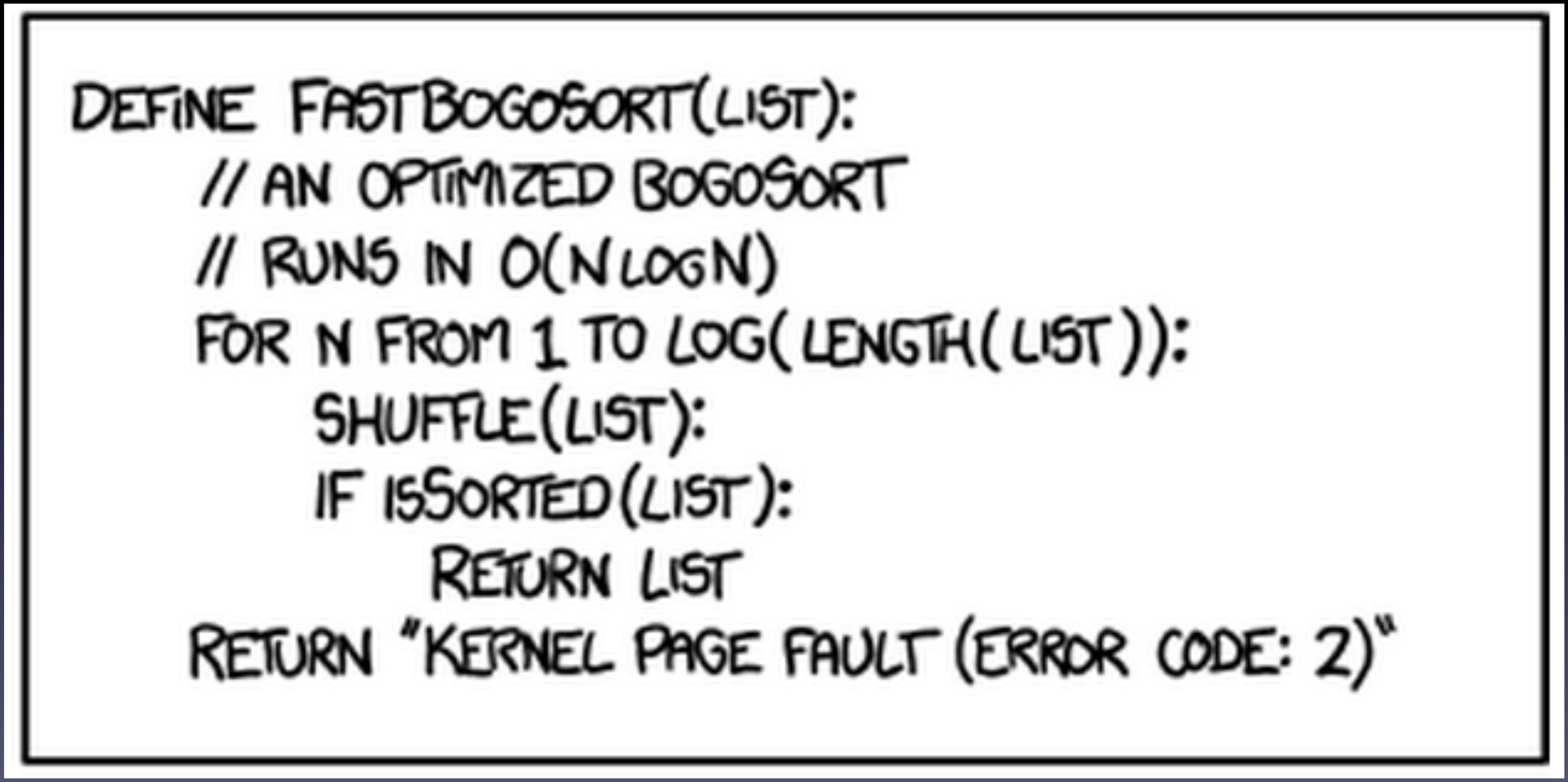
Awesome Big Data Algorithms



```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

http://xkcd.com/1185/

Awesome Big Data Algorithms

C. Titus Brown

ctb@msu.edu

Asst Professor, Michigan State University

(Microbiology, Computer Science, and BEACON)

Welcome!

- More of a computational scientist than a computer scientist; will be using *simulations* to demo & explore algorithm behavior.

- Send me questions/comments @ctitusbrown, or ctb@msu.edu.

“Features”

- I will be using Python rather than C++, because Python is easier to read.
- I will be using IPython Notebook to demo.
- I apologize in advance for not covering *your* favorite data structure or algorithm.

Outline

- The basic idea
- Three examples
  - Skip lists (a fast key/value store)
  - HyperLogLog Counting (counting discrete elements)
  - Bloom filters and CountMin Sketches
- Folding, spindling, and mutilating DNA sequence
- References and further reading

The basic idea

- Problem: you have a **lot** of data to count, track, or otherwise analyze.
- This data is Data of Unusual Size, i.e. you can't just brute force the analysis.
- For example,
  - Count the approximate number of distinct elements in a very large (infinite?) data set
  - Optimize queries by using an efficient but approximate prefilter
  - Determine the frequency distribution of distinct elements in a very large data set.

Online and streaming vs. offline

"Large is hard; infinite is much easier."

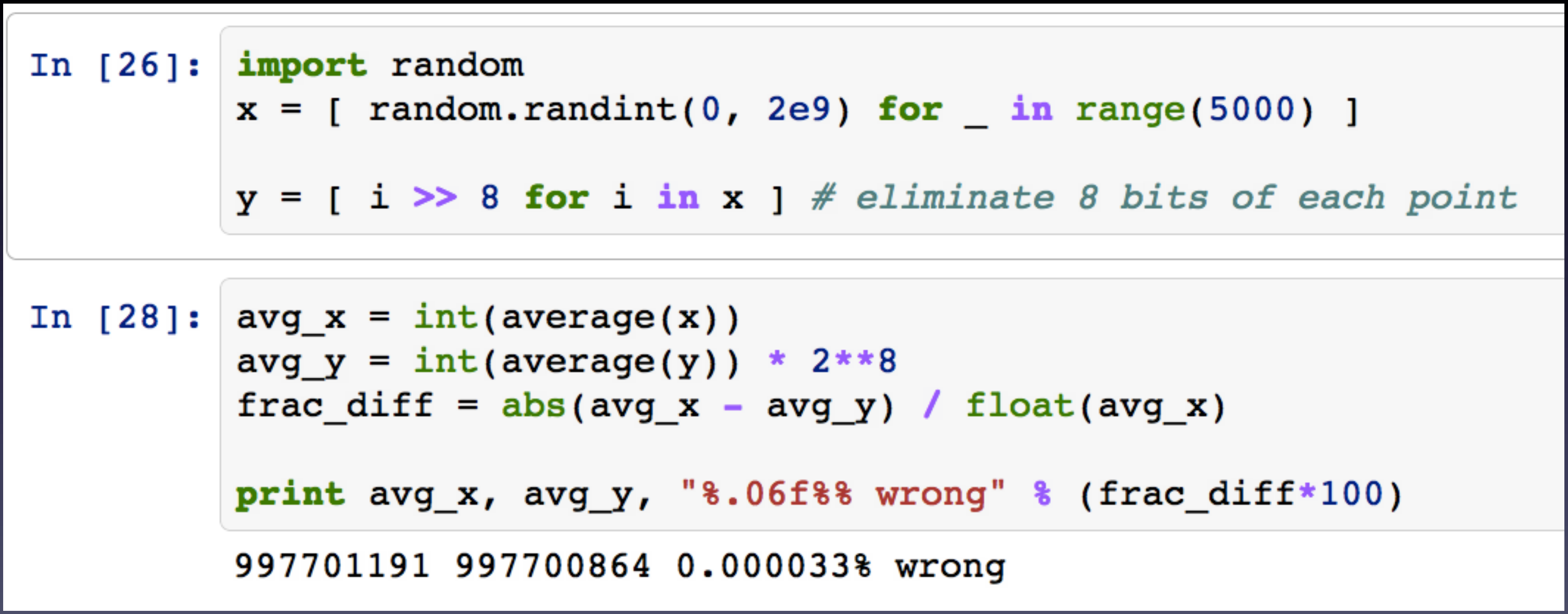
- *Offline* algorithms analyze an entire data set all at once.
- *Online* algorithms analyze data serially, one piece at a time.
- *Streaming* algorithms are online algorithms that can be used for very memory & compute limited analysis.

Exact vs *random* or *probabilistic*

- Often an approximate answer is sufficient, esp if you can place bounds on how wrong the approximation is likely to be.
- Often random algorithms or probabilistic data structures can be found with good *typical* behavior but bad *worst case* behavior.

For one (stupid) example

You can trim 8 bits off of integers for the purpose of averaging them



```
In [26]: import random
         x = [ random.randint(0, 2e9) for _ in range(5000) ]

         y = [ i >> 8 for i in x ] # eliminate 8 bits of each point
```

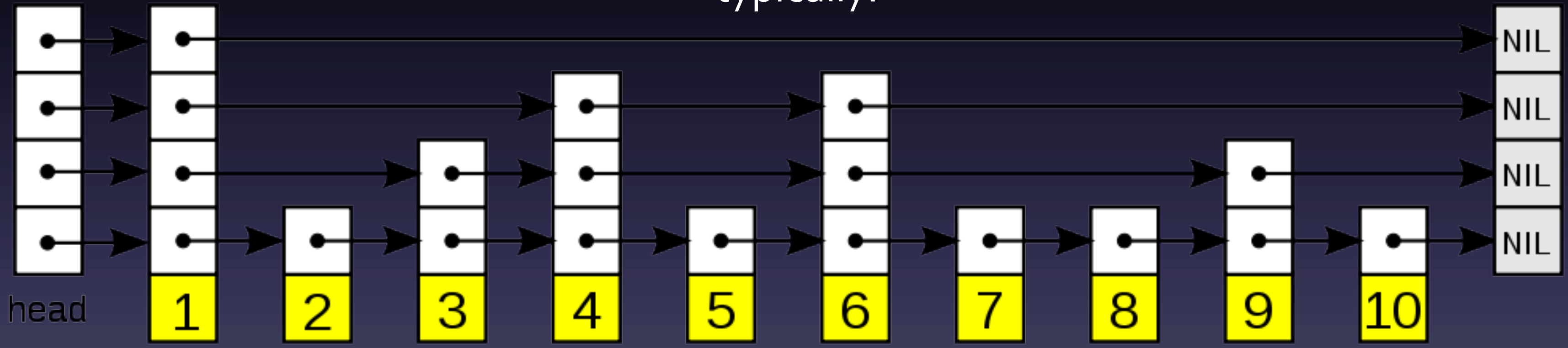
```
In [28]: avg_x = int(average(x))
         avg_y = int(average(y)) * 2**8
         frac_diff = abs(avg_x - avg_y) / float(avg_x)

         print avg_x, avg_y, "%.06f%% wrong" % (frac_diff*100)
```

```
997701191 997700864 0.000033% wrong
```

Skip lists

A *randomly indexed* improvement on linked lists.

Each node can belong to one or more vertical “levels”, which allow fast search/insertion/deletion – ~O(log(n)) typically!

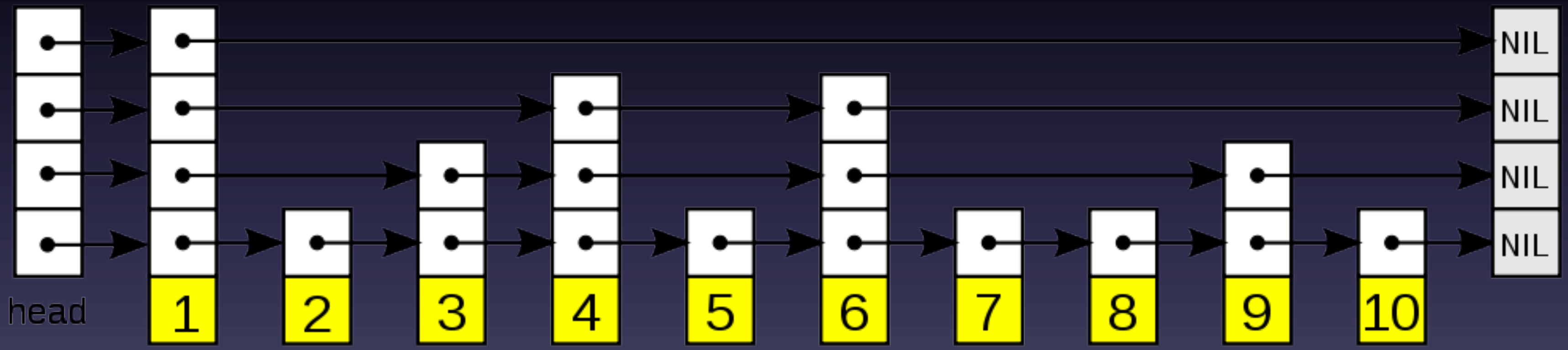


wikipedia

Skip lists

A *randomly indexed* improvement on linked lists.

Very easy to implement; asymptotically good behavior.



From reddit, "if someone held a gun to my head and asked me to implement an efficient set/map storage, I would implement a skip list."

(Response: "does this happen to you a lot??")

wikipedia

Channel randomness!

- If you can construct or rely on randomness, then you can easily get good typical behavior.
- Note, a *good hash function* is essentially the same as a good random number generator…

HyperLogLog cardinality counting

- Suppose you have an incoming stream of many, many “objects”.
- And you want to track how many distinct items there are, and you want to accumulate the count of distinct objects over time.

Relevant digression:

- Flip some unknown number of coins. Q: what is something simple to track that will tell you roughly how many coins you've flipped?

- A: longest run of HEADs. Long runs are very rare and are correlated with how many coins you've flipped.

Cardinality counting with HyperLogLog

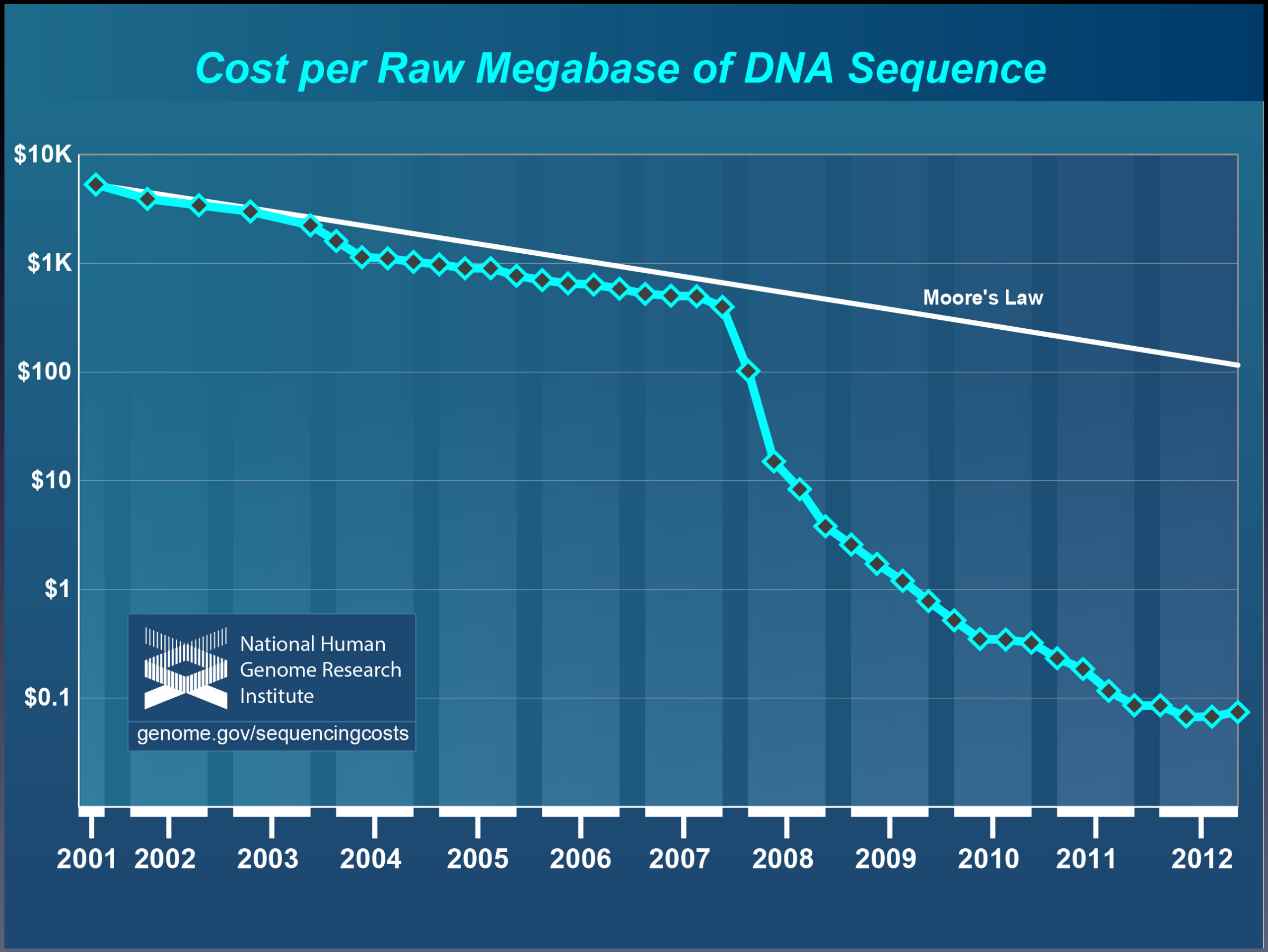
- Essentially, use longest run of 0-bits observed in a hash value.
- Use multiple hash functions so that you can take the average.
- Take harmonic mean + low/high sampling adjustment => result.

Bloom filters

- A set membership data structure that is probabilistic but only yields false positives.
- Trivial to implement; hash function is main cost; extremely memory efficient.

My research applications

Biology is fast becoming a data-driven science.



http://www.genome.gov/sequencingcosts/

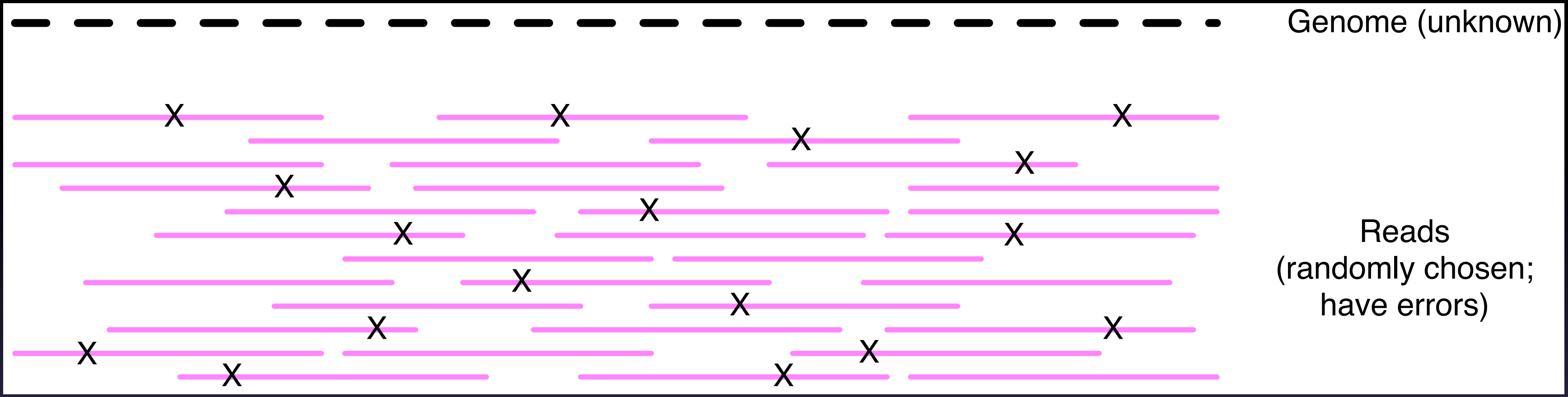
Shotgun sequencing analogy:
feeding books into a paper shredder, digitizing the shreds, and reconstructing the book.

Although for books, we often know the language and not just the alphabet ☺

Shotgun sequencing is --

- Randomly ordered.
- Randomly sampled.
- Too big to efficiently do multiple passes

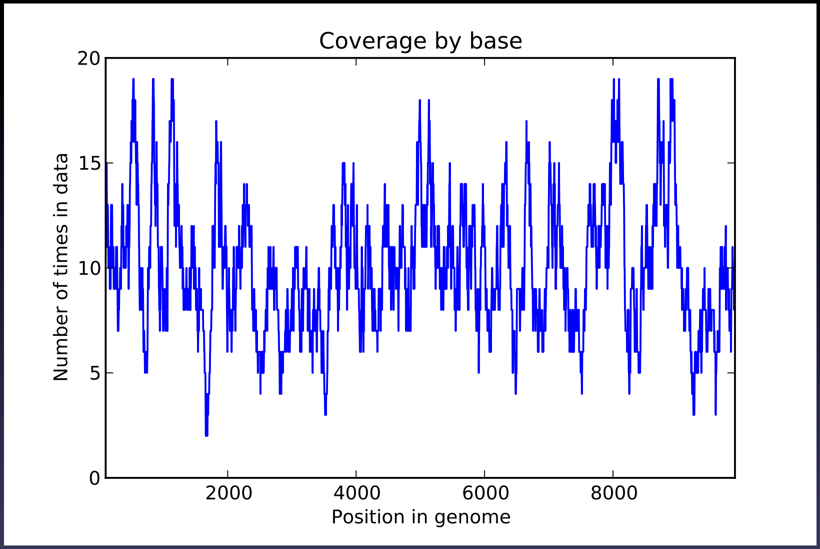
Shotgun sequencing

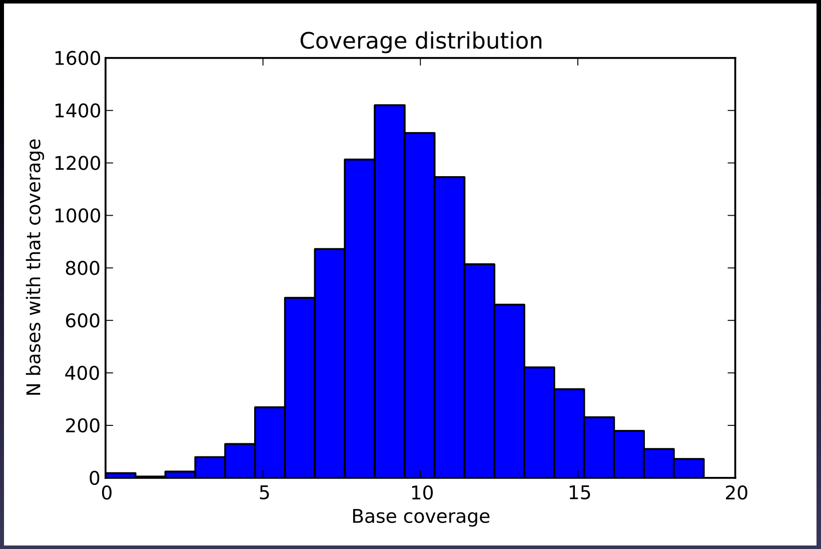


"Coverage" is simply the average number of reads that overlap each true base in genome.

Here, the coverage is ~10 – just draw a line straight down from the top through all of the reads.

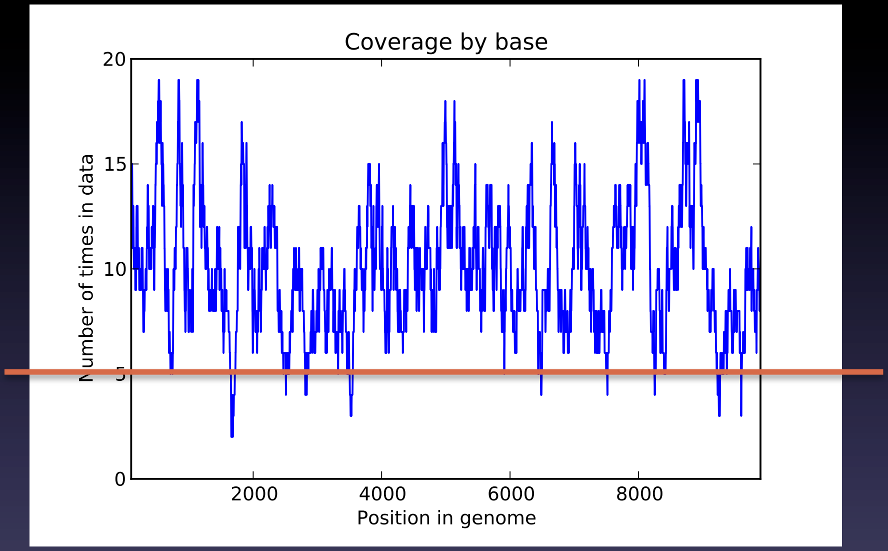
Random sampling => deep sampling needed

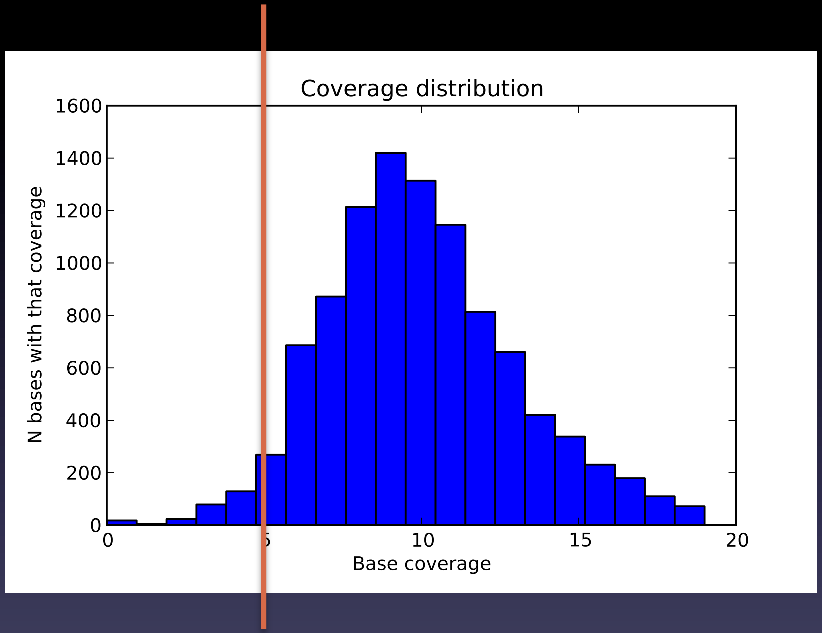




Typically 10-100x needed for robust recovery (300 Gbp for human)

Random sampling => deep sampling needed

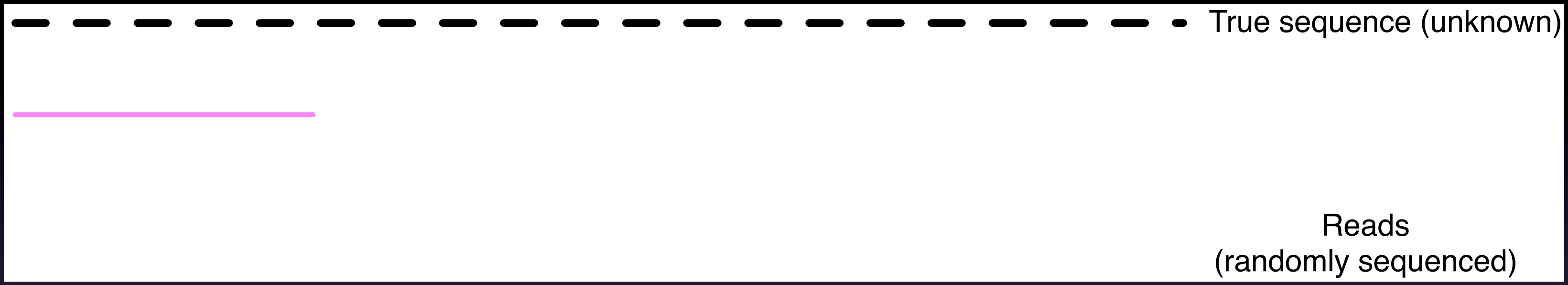




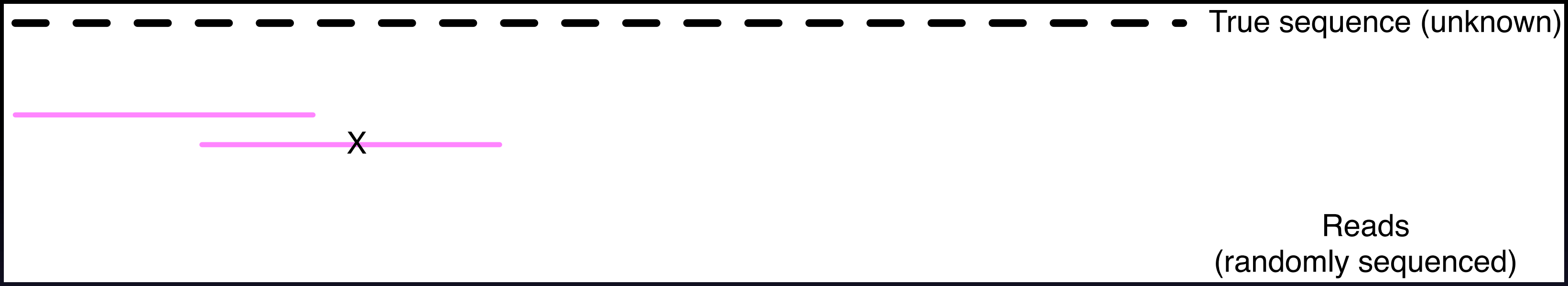
Typically 10-100x needed for robust recovery (300 Gbp for human)

But this data is massively redundant!! Only need 5x *systematic!*
All the stuff above the red line is unnecessary!

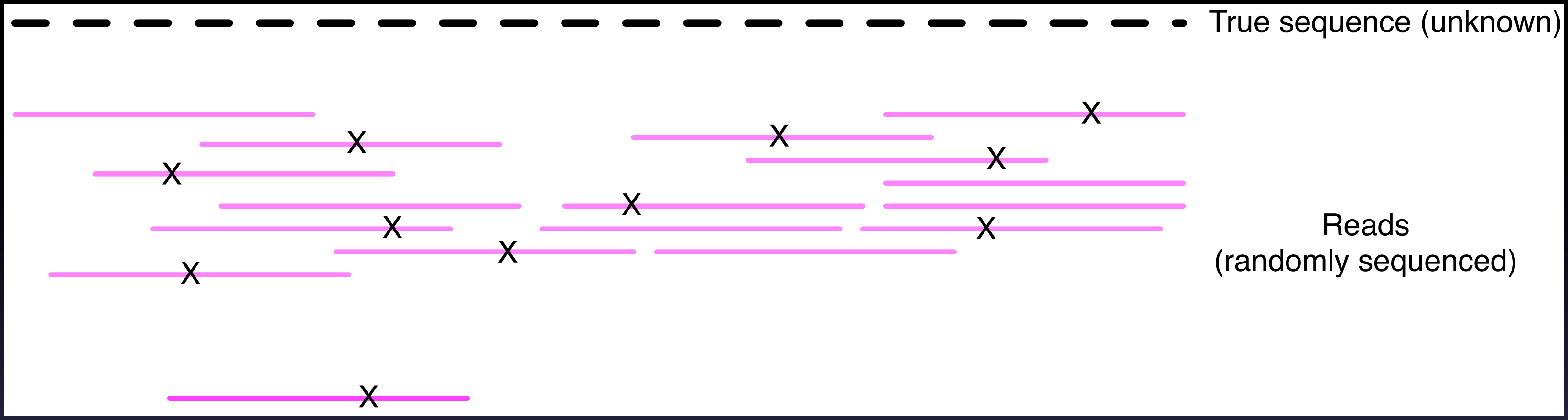
Streaming algorithm to do so: digital normalization



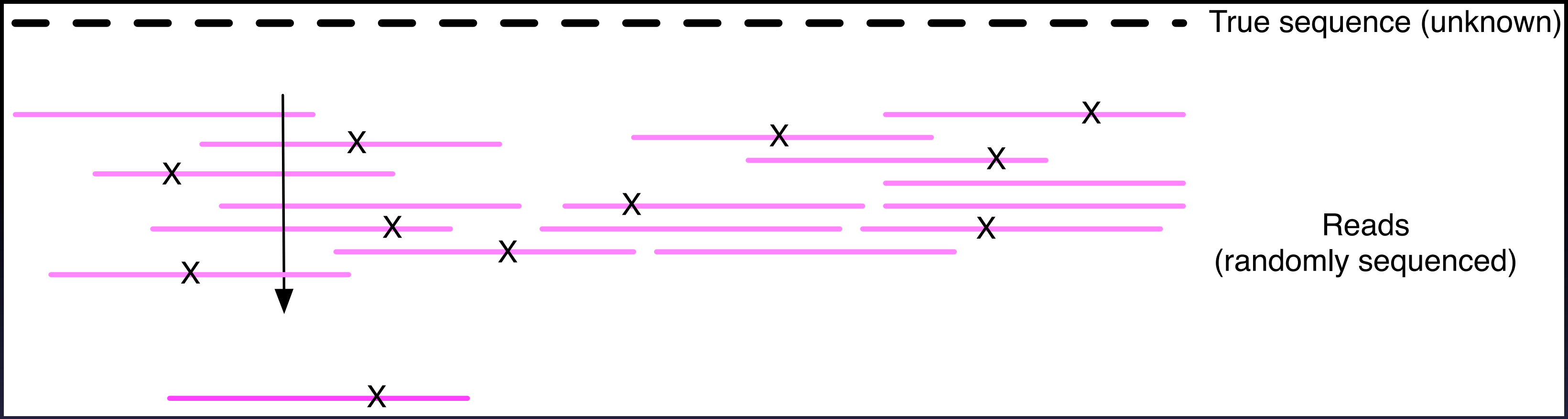
Digital normalization



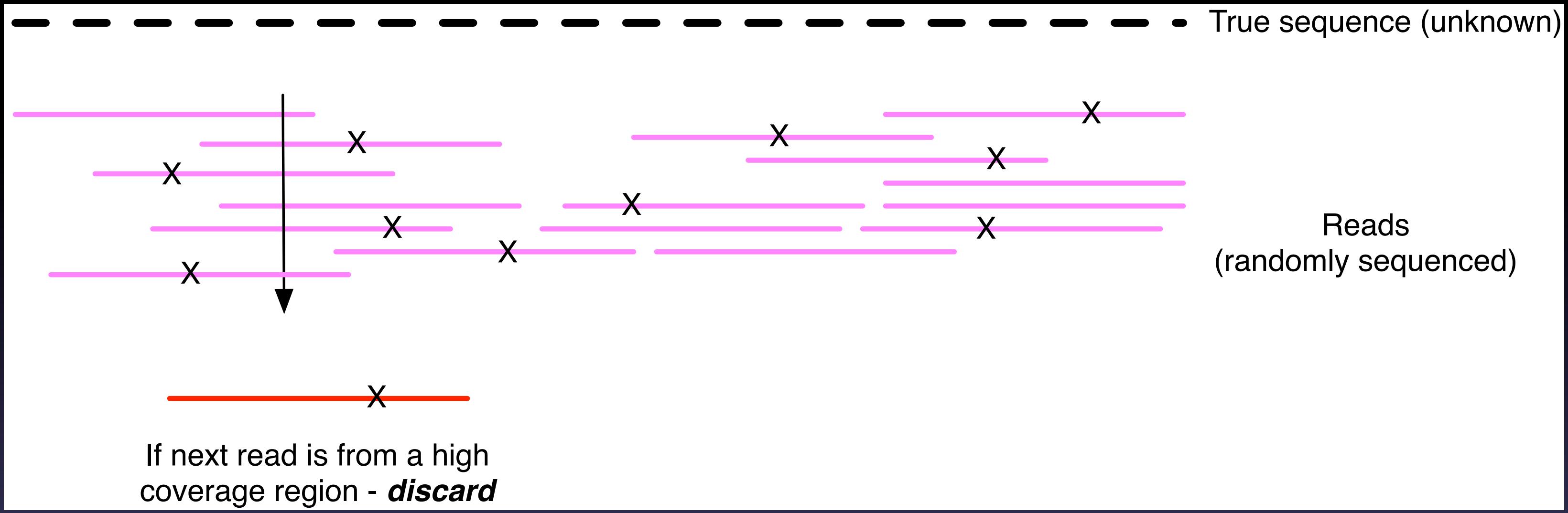
Digital normalization



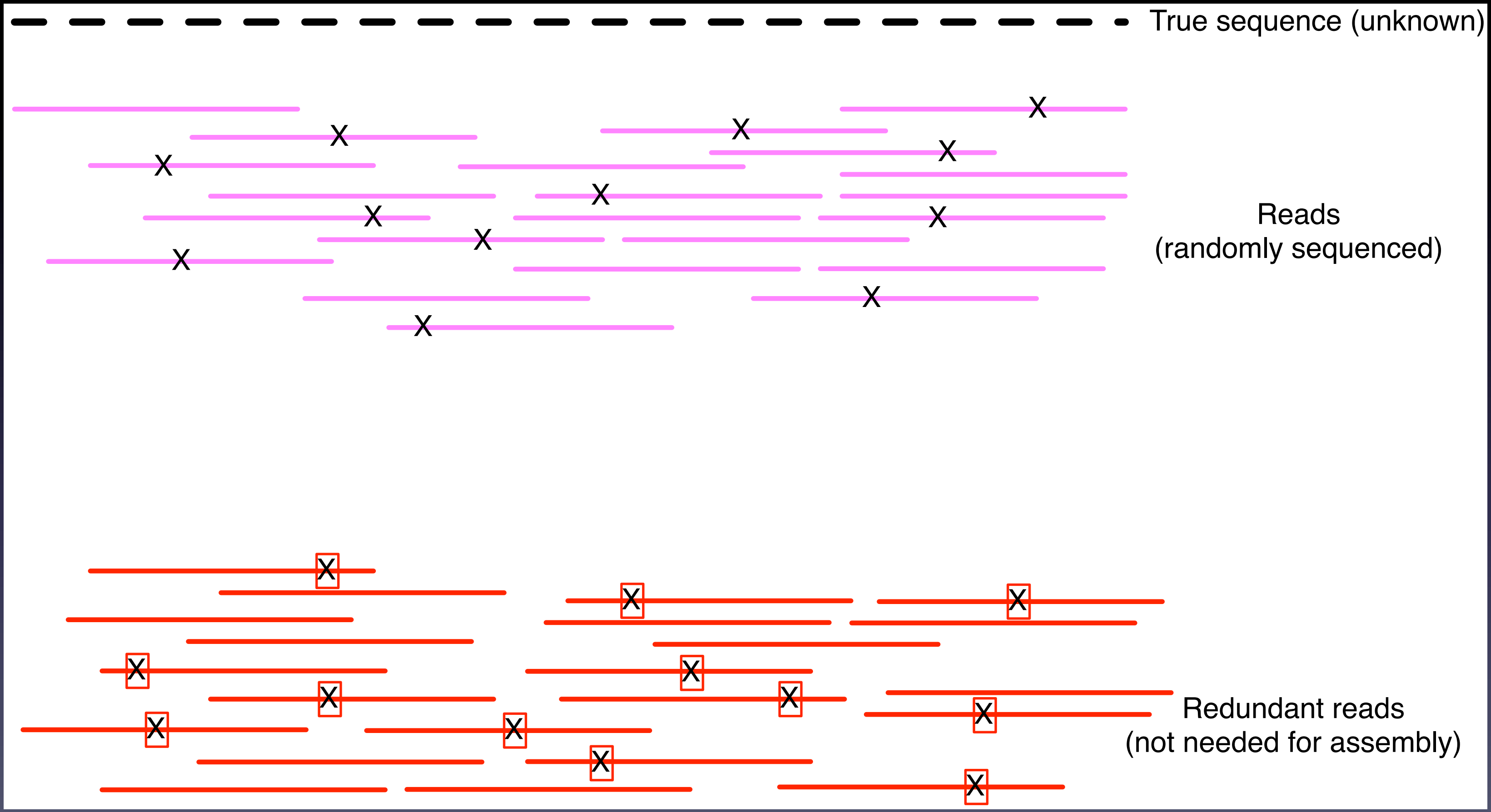
Digital normalization



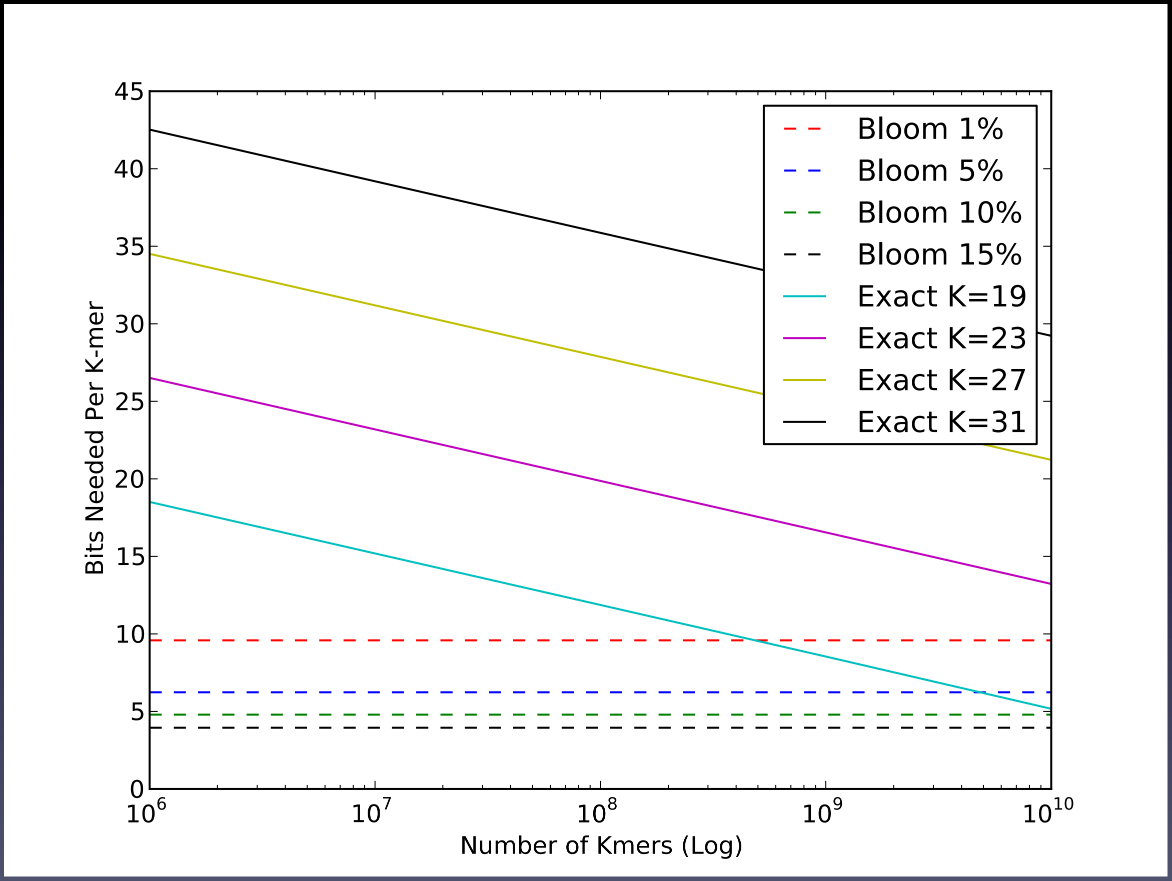
Digital normalization



Digital normalization



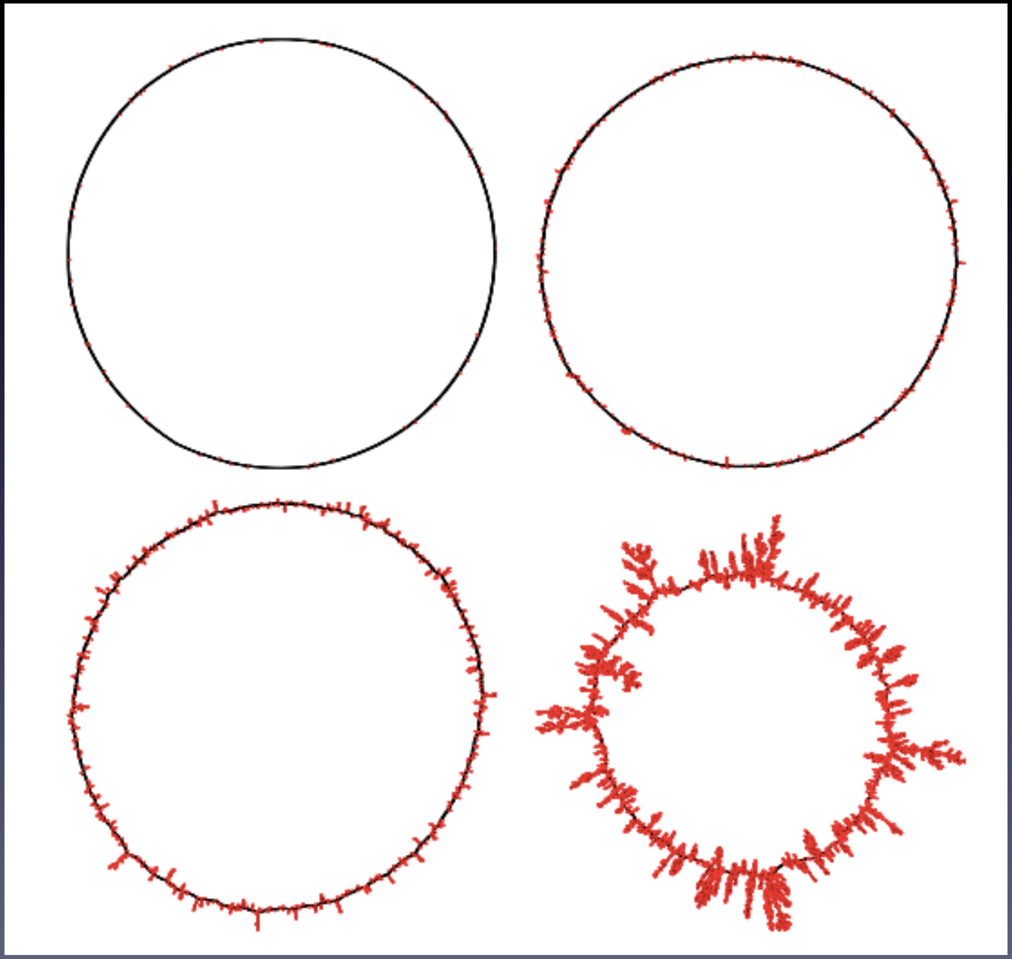
Storing data this way is better than best-possible information-theoretic storage.



Pell et al., PNAS 2012

Use Bloom filter to store graphs

Graphs only *gain* nodes because of Bloom filter false positives.



Pell et al., PNAS 2012

Some assembly details

- This *was* completely intractable.
- Implemented in C++ and Python; “good practice” (?)
- We’ve changed scaling behavior from *data* to *information*.
- Practical scaling for ~soil metagenomics is 10x:
  - need < 1 TB of RAM for ~2 TB of data, ~2 weeks.
  - Before, ~10TB.
- Smaller problems are pretty much solved.
- Just beginning to explore threading, multicore, etc. (BIG DATA grant proposal)
- Goal is to scale to 50 Tbp of data (~5-50 TB RAM currently)

Concluding thoughts

- Channel randomness.
- Embrace streaming.
- Live with minor uncertainty.
- Don't be afraid to discard data.

(Also, I'm an open source hacker who can confer PhDs, in exchange for long years of low pay living in Michigan. E-mail me! And don't talk to Brett Cannon about PhDs first.)

References

SkipLists: Wikipedia, and John Shipman's code:

http://infohost.nmt.edu/tcc/help/lang/python/examples/pyskip/pyskip.pdf

HyperLogLog: Aggregate Knowledge's blog,

http://blog.aggregateknowledge.com/2012/10/25/sketch-of-the-day-hyperloglog-cornerstone-of-a-big-data-infrastructure/

And: https://github.com/svpcom/hyperloglog

Bloom Filters: Wikipedia

Our work: http://ivory.idyll.org/blog/ and http://ged.msu.edu/interests.html

ctb@msu.edu