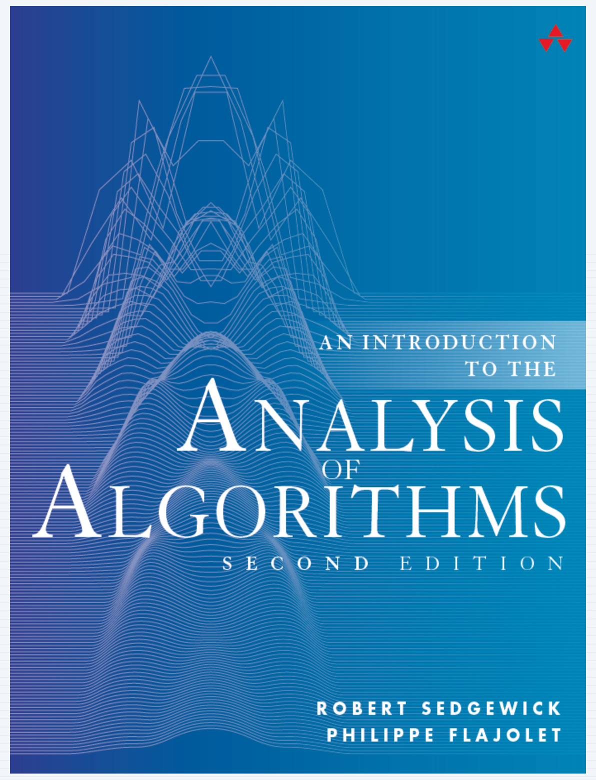
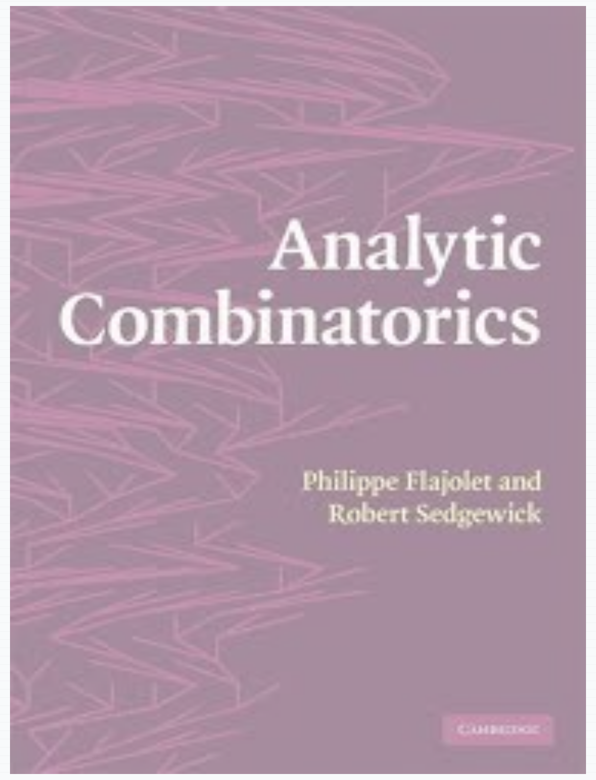


Analysis of Algorithms

Original MOOC title: ANALYTIC COMBINATORICS, PART ONE



Analytic Combinatorics

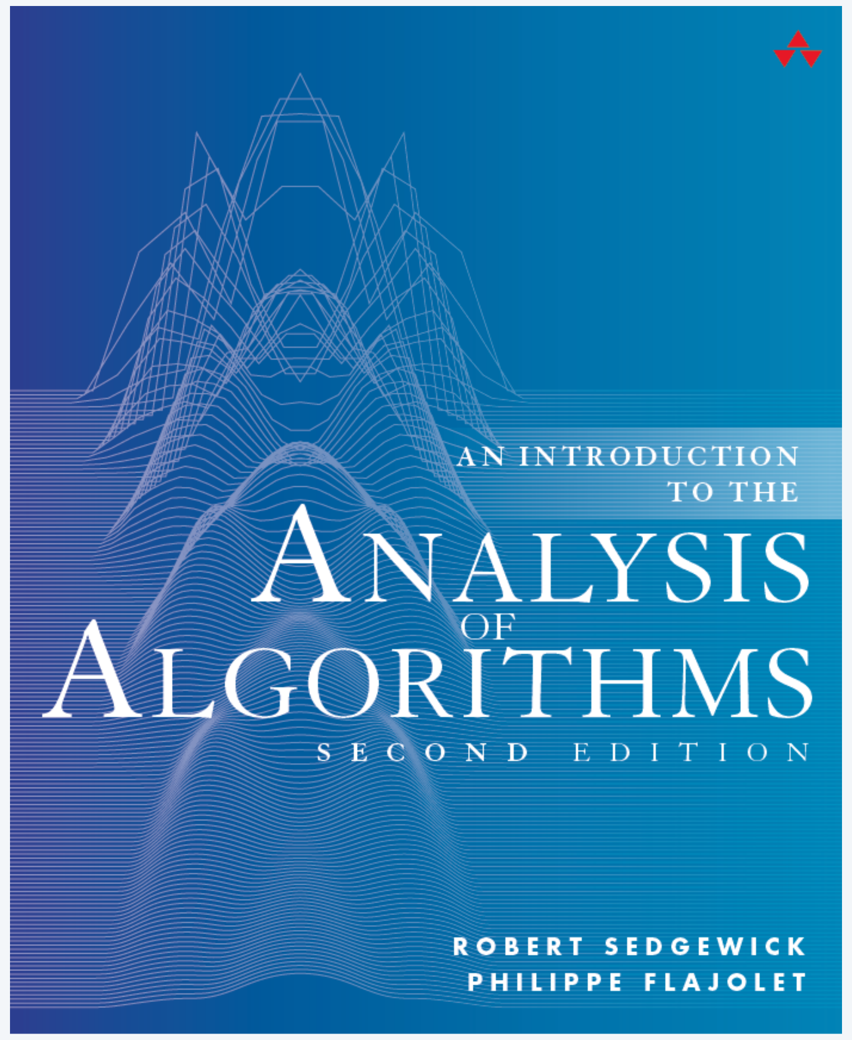
Original MOOC title: ANALYTIC COMBINATORICS, PART TWO

http://aofa.cs.princeton.edu
http://ac.cs.princeton.edu

Overview

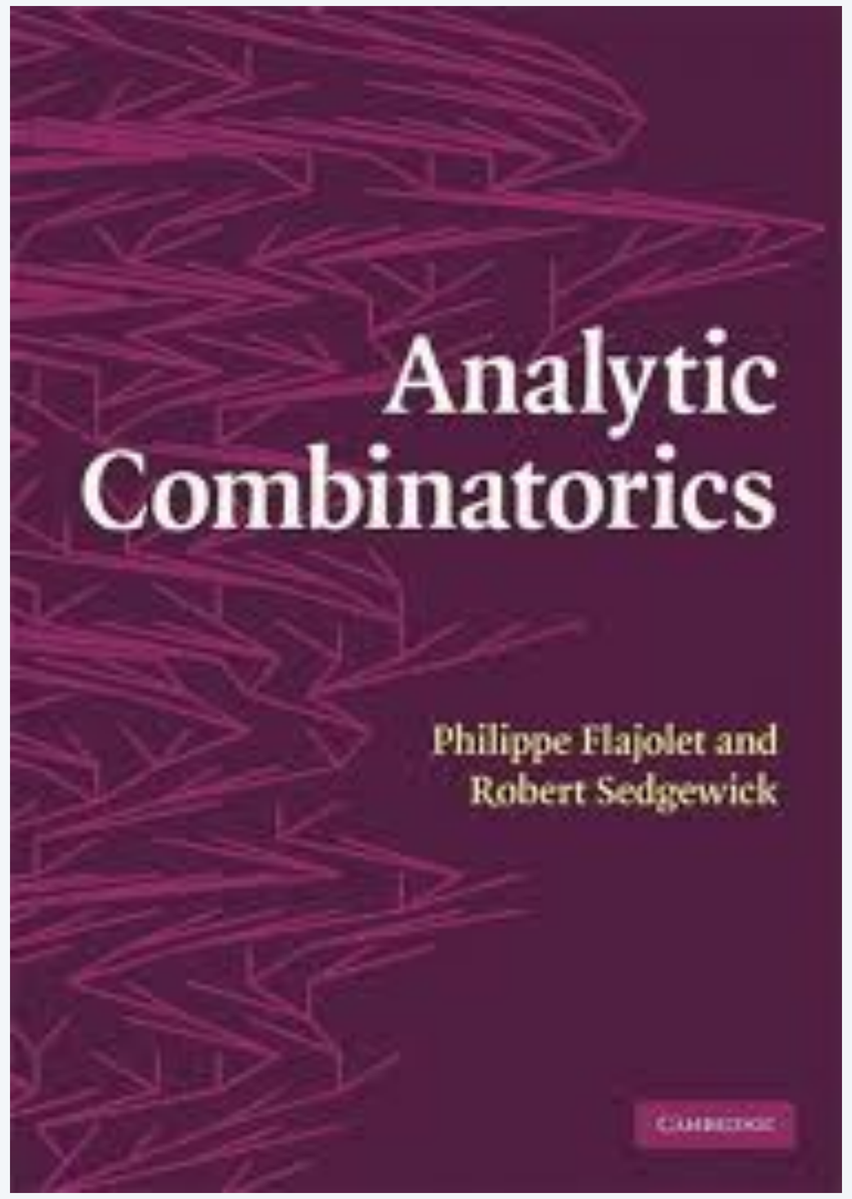
Analysis of algorithms

- Methods and models for the analysis of algorithms.
- Basis for a scientific approach.
- Mathematical methods from classical analysis.
- Combinatorial structures and associated algorithms.



Analytic combinatorics

- Study of properties of large combinatorial structures.
- A foundation for analysis of algorithms, but widely applicable.
- Symbolic method for encapsulating precise description.
- Complex analysis to extract useful information.

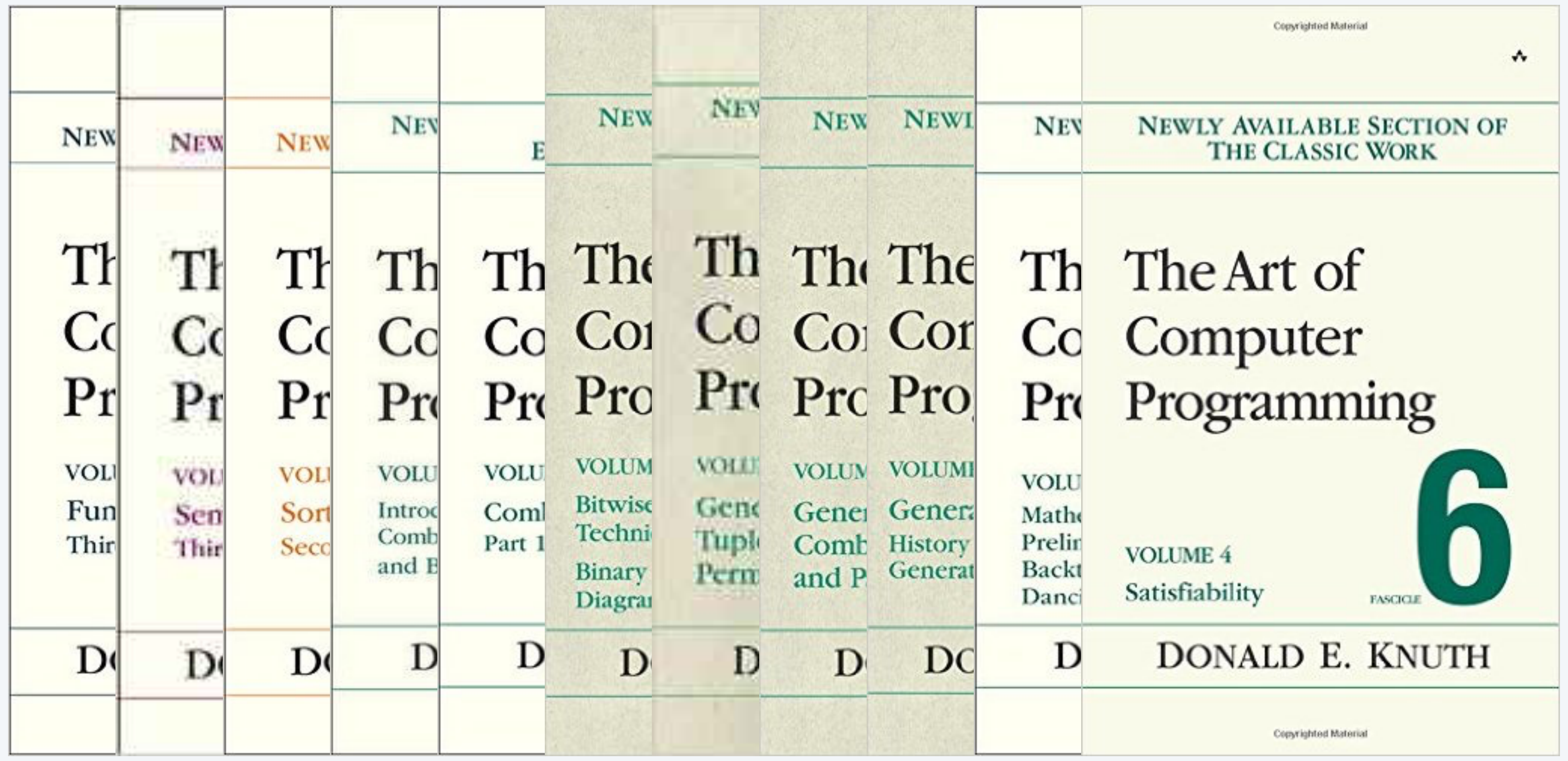


Are these courses for me?

Sure, if you can answer "yes" to these questions.

- Do you like to program?
- Do you like math?
- Have you studied *Algorithms*?

- Would you like to be able to read Knuth's books?



- and Flajolet's papers?

Q. Why study the analysis of algorithms and analytic combinatorics?

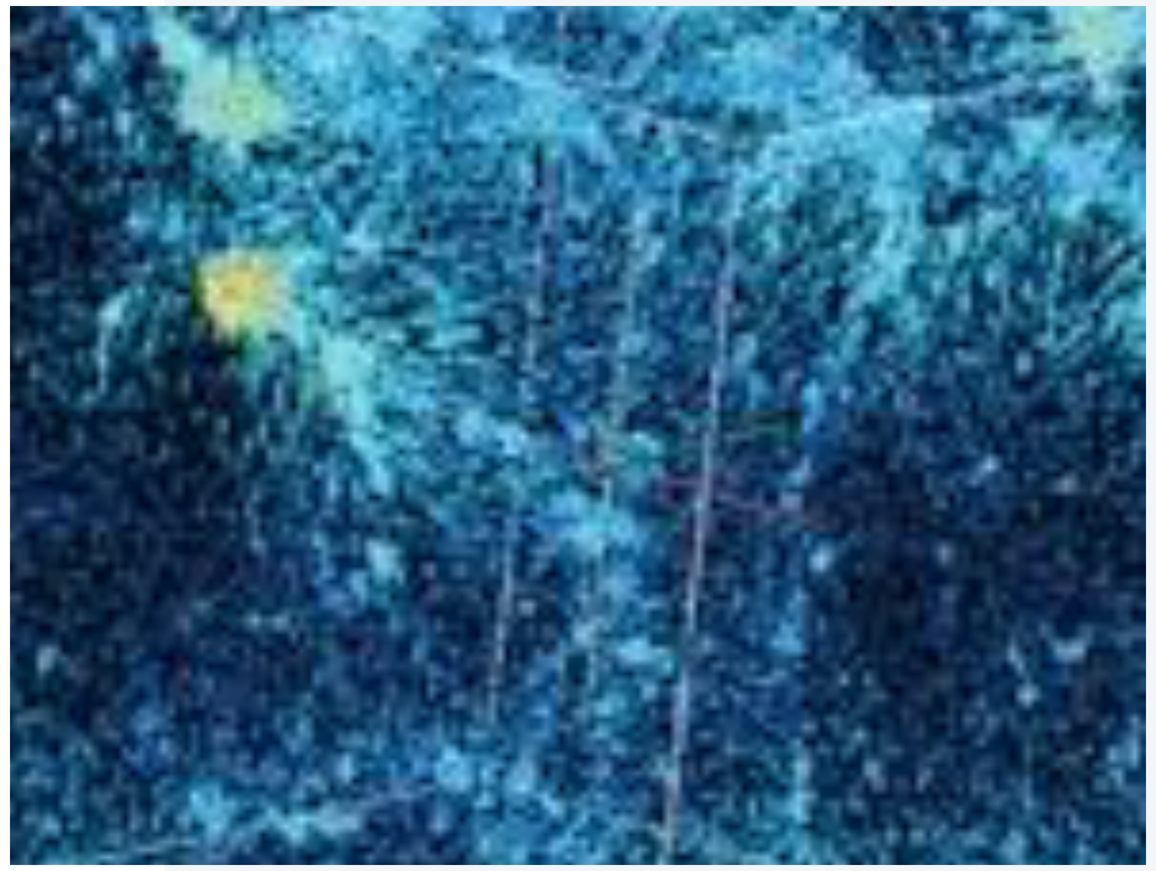
A. For many of the same reasons we study *algorithms* (next)!

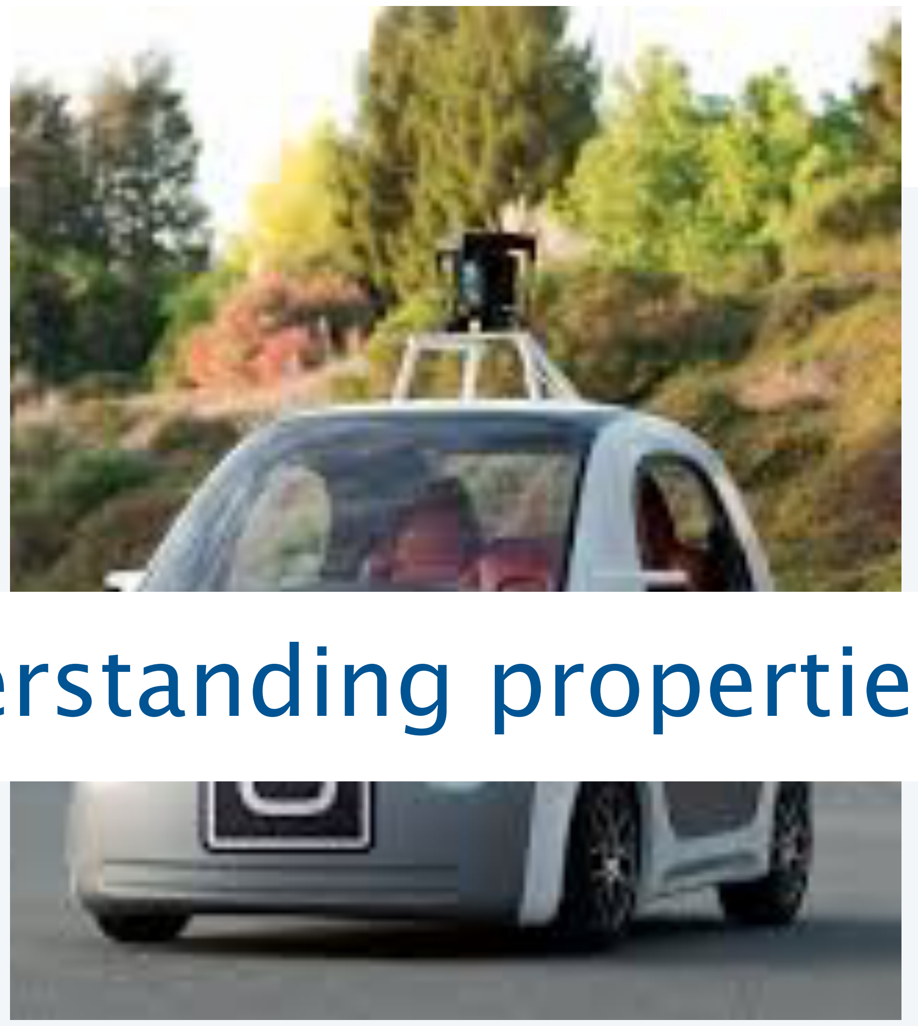
Why study the analysis of algorithms and analytic combinatorics?

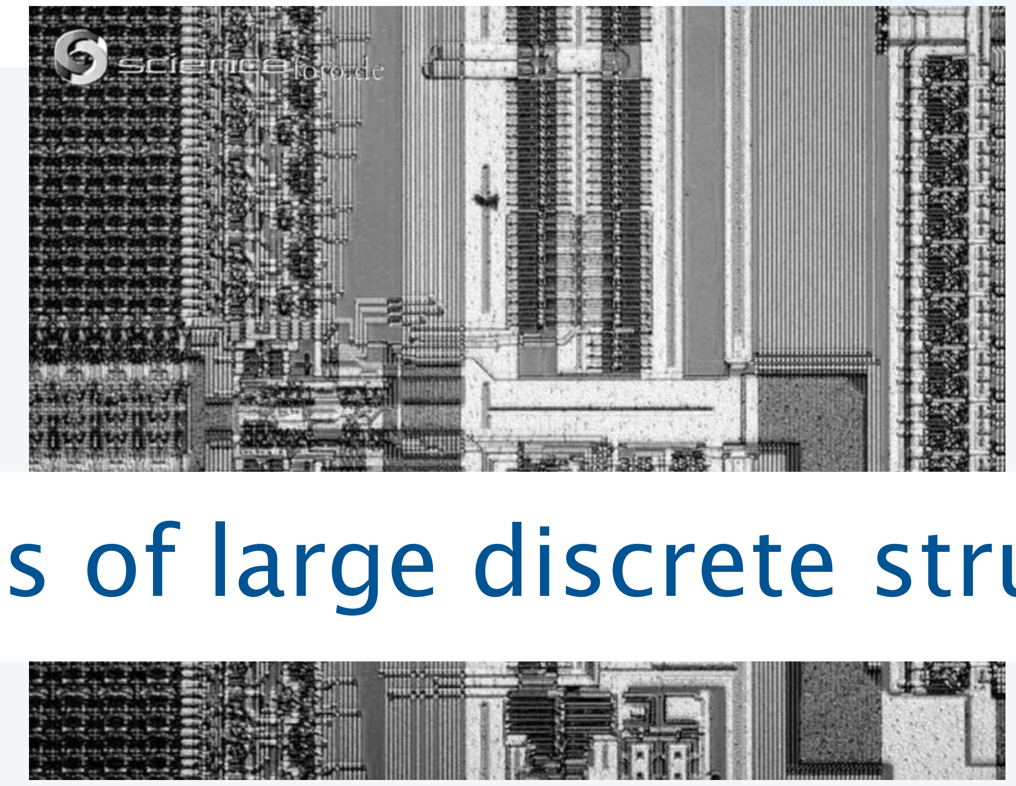
Their impact is broad and far-reaching.

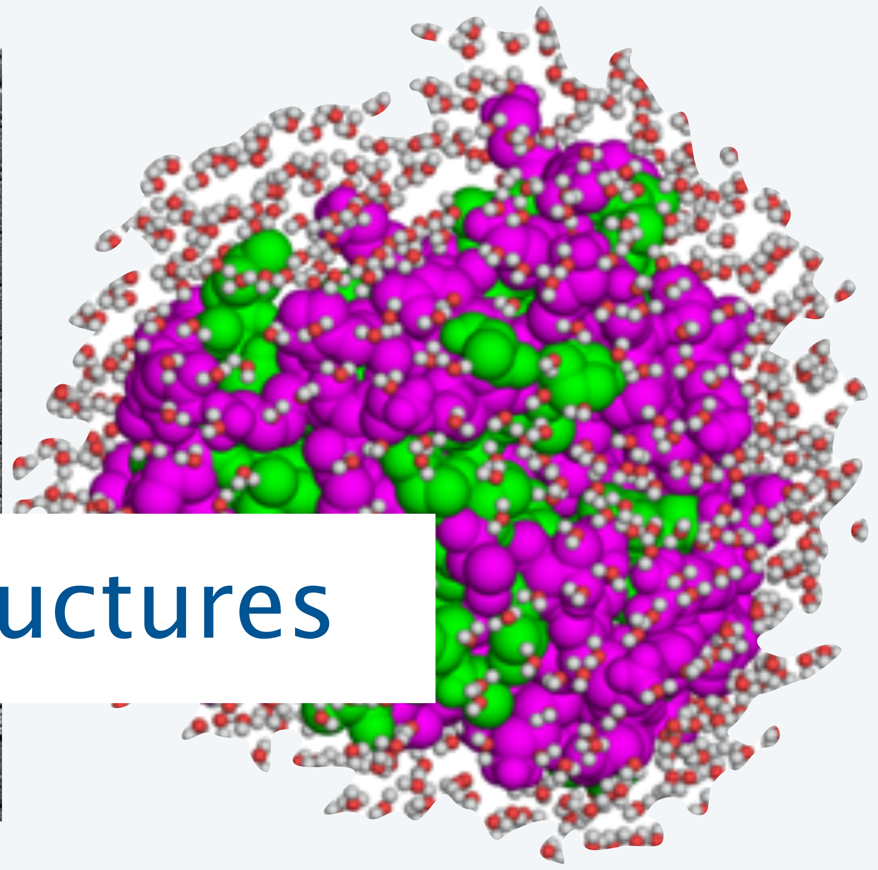
Internet. Web search, packet routing, file sharing, ...
Biology. Human genome project, protein folding, ...
Computer design. Circuit layout, file system, compilers, ...
Multimedia. Movies, video games, virtual reality, ...
Security. Cell phones, e-commerce, voting machines, ...
Social networks. Recommendations, news feeds, advertisements, ...
Physics. N-body simulation, particle collision simulation, ...
Big data. Deep learning, autonomous vehicles, ...

⋮









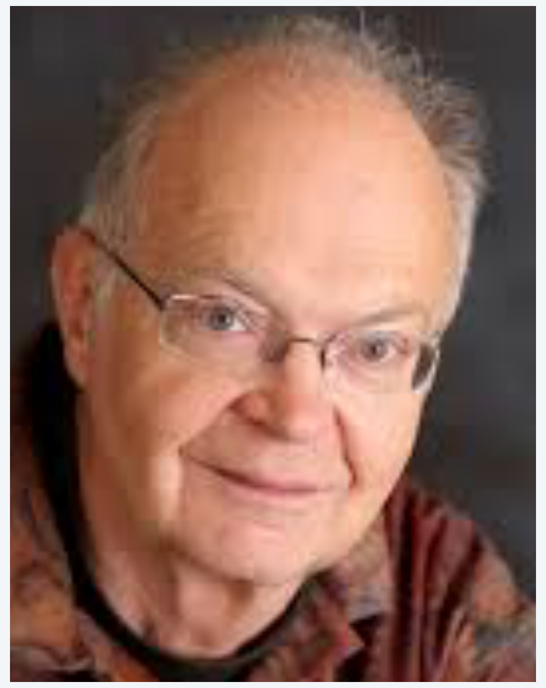
All involve understanding properties of large discrete structures

Why study the analysis of algorithms and analytic combinatorics?

Old roots, new opportunities.

- Analysis of algorithms dates at least to Euclid.
- Practiced by Turing and von Neumann in 1940s.
- Mostly developed by Knuth starting in 1960s.
- Steady evolution for decades.
- Analytic combinatorics dates to Euler and earlier.
- Mostly developed by Flajolet starting in 1980s.
- *Many algorithms are waiting to be understood.*
- *Many theorems are waiting to be discovered.*

"father of analysis of algorithms"



Don Knuth

"father of analytic combinatorics"

Philippe Flajolet

" If I have seen further, it is by standing on the shoulders of giants."

– Isaac Newton

Why study the analysis of algorithms and analytic combinatorics?

To solve problems that could not otherwise be addressed.

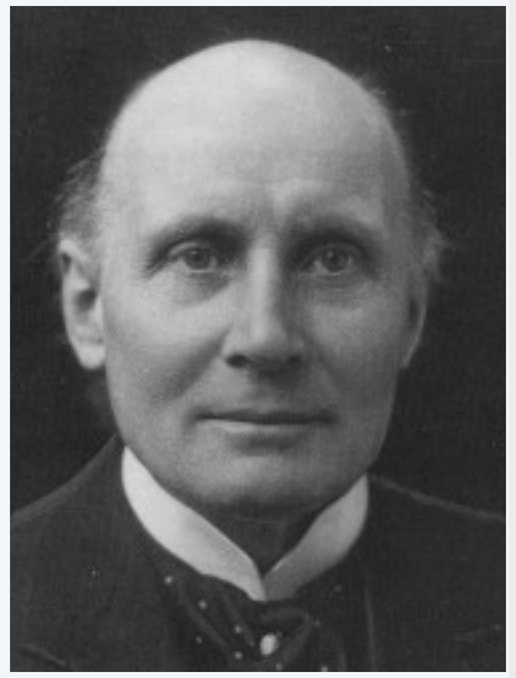
Example: Cardinality estimation (stay tuned).

```
pool-71-104-94-246.lsanca.dsl-w.verizon.net
117.222.48.163
pool-71-104-94-246.lsanca.dsl-w.verizon.net
1.23.193.58
188.134.45.71
1.23.193.58
gsearch.CS.Princeton.EDU
pool-71-104-94-246.lsanca.dsl-w.verizon.net
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
CPE-121-218-151-176.lnse3.cht.bigpond.net.au
117.211.88.36
msnbot-131-253-46-251.search.msn.com
msnbot-131-253-46-251.search.msn.com
pool-71-104-94-246.lsanca.dsl-w.verizon.net
gsearch.CS.Princeton.EDU
CPE001cdfbc55ac-CM0011ae926e6c.cpe.net.cable.rogers.com
CPE001cdfbc55ac-CM0011ae926e6c.cpe.net.cable.rogers.com
118-171-27-8.dynamic.hinet.net
cpe-76-170-182-222.socal.res.rr.com
```

← How many of these are different?

Why study the analysis of algorithms and analytic combinatorics?

For intellectual stimulation.



" The point of mathematics is that in it we have always got rid of the particular instance, … no mathematical truths apply merely to fish, or merely to stones, or merely to colours. So long as you are dealing with pure mathematics, you are in the realm of complete and absolute abstraction. … Mathematics is thought moving in the sphere of complete abstraction from any particular instance of what it is talking about.

– Alfred North Whitehead

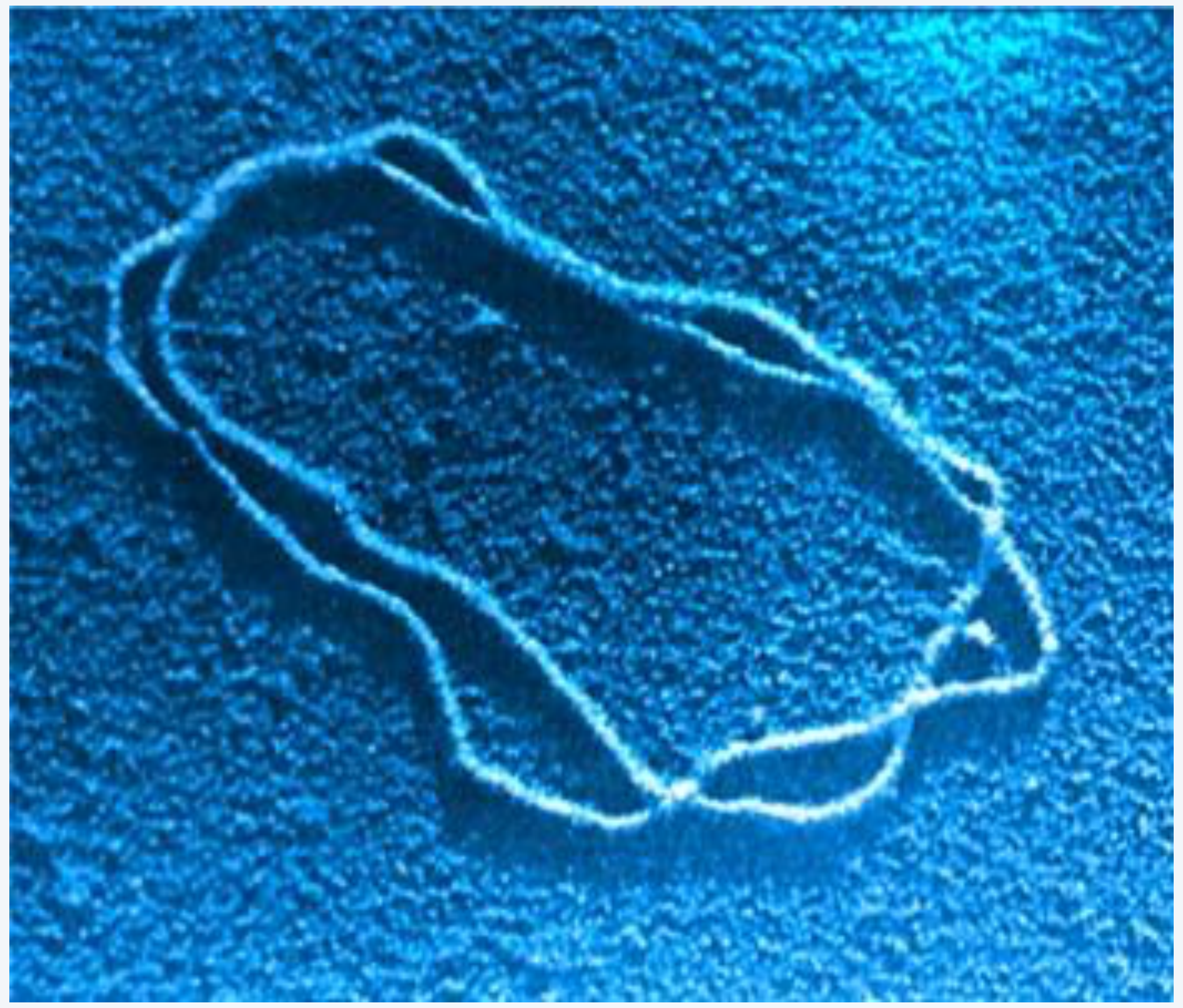
Abstract Thought 379, by Theo Dapore

" Here's to pure mathematics—may it never be of any use to anybody."

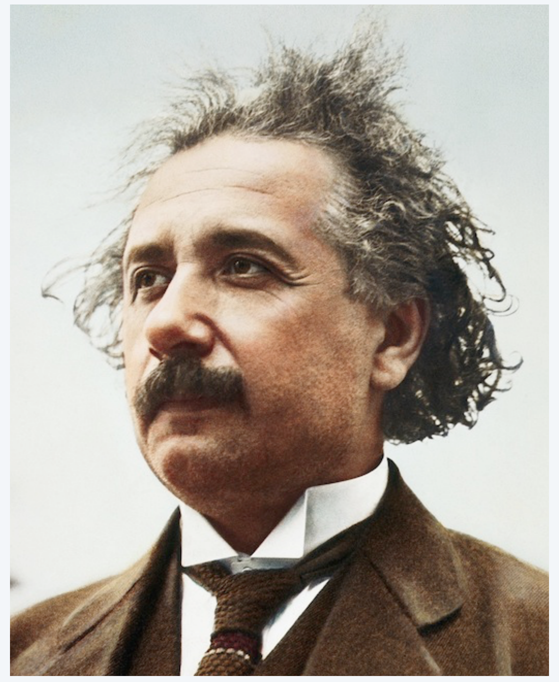
– attributed to G. H. Hardy

Why study the analysis of algorithms and analytic combinatorics?

They may unlock the secrets of life and of the universe.



" Pure mathematics is, in its way, the poetry of logical ideas. One seeks the most general ideas of operation which will bring together in simple, logical and unified form the largest possible circle of formal relationships. In this effort toward logical beauty spiritual formulas are discovered necessary for the deeper penetration into the laws of nature."



– Albert Einstein

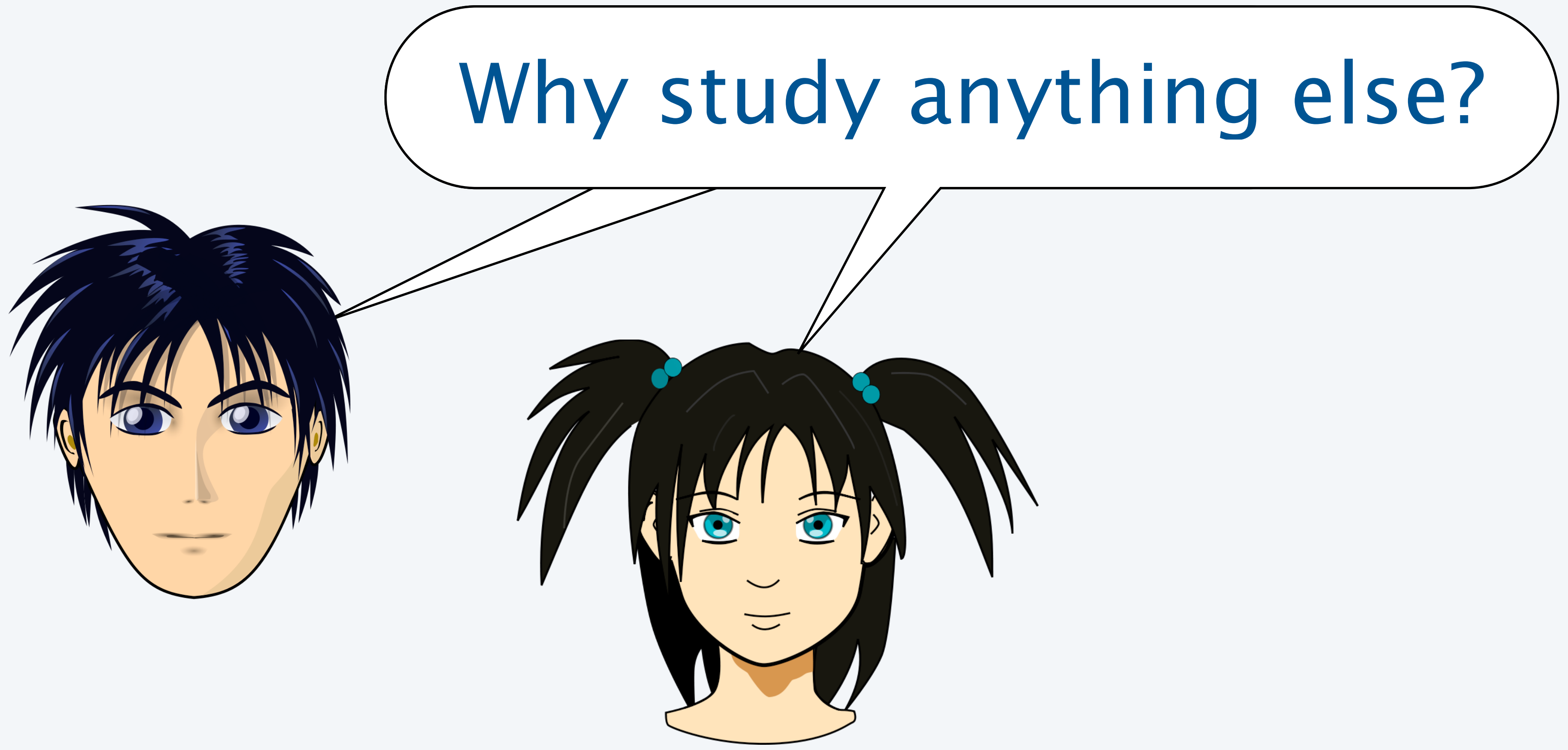
Why study the analysis of algorithms and analytic combinatorics?

For fun and profit.

Why study the analysis of algorithms and analytic combinatorics?

Some compelling reasons

- Their impact is broad and far-reaching.
- Old roots, new opportunities.
- To solve problems that could not otherwise be addressed.
- For intellectual stimulation.
- They may unlock the secrets of life and of the universe.
- For fun and profit.



This lecture. A case in point.

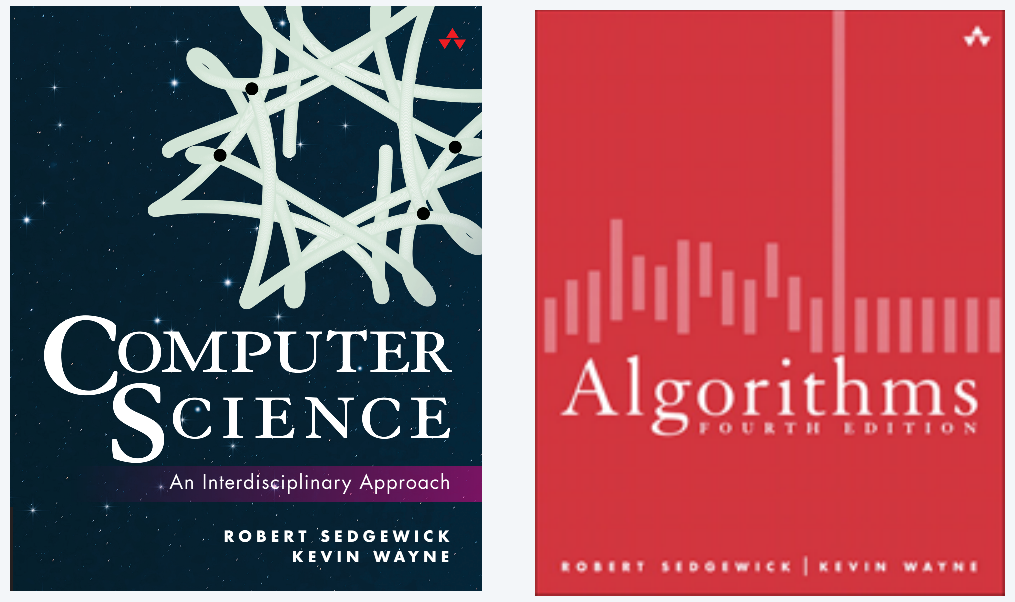
Context for this lecture

Purpose. Prepare for the study of the analysis of algorithms *in the context of an important application.*

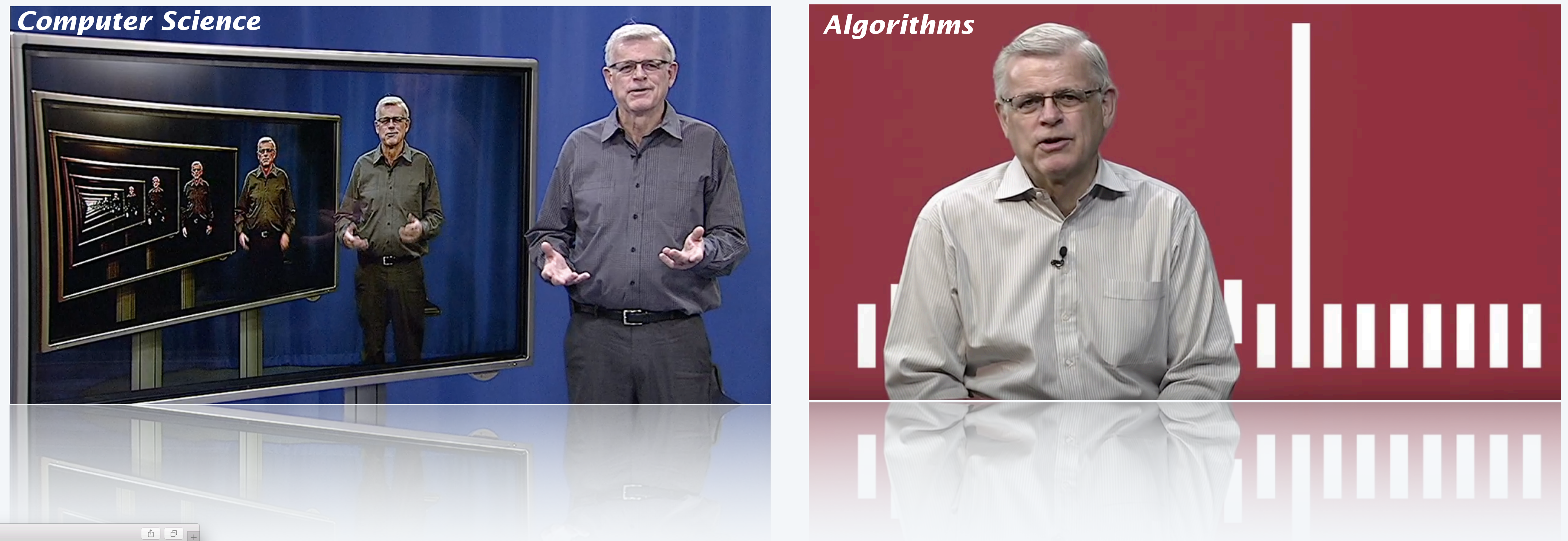
Assumed. Familiarity with undergraduate-level Java programming, computer science, and algorithms.

For reference.

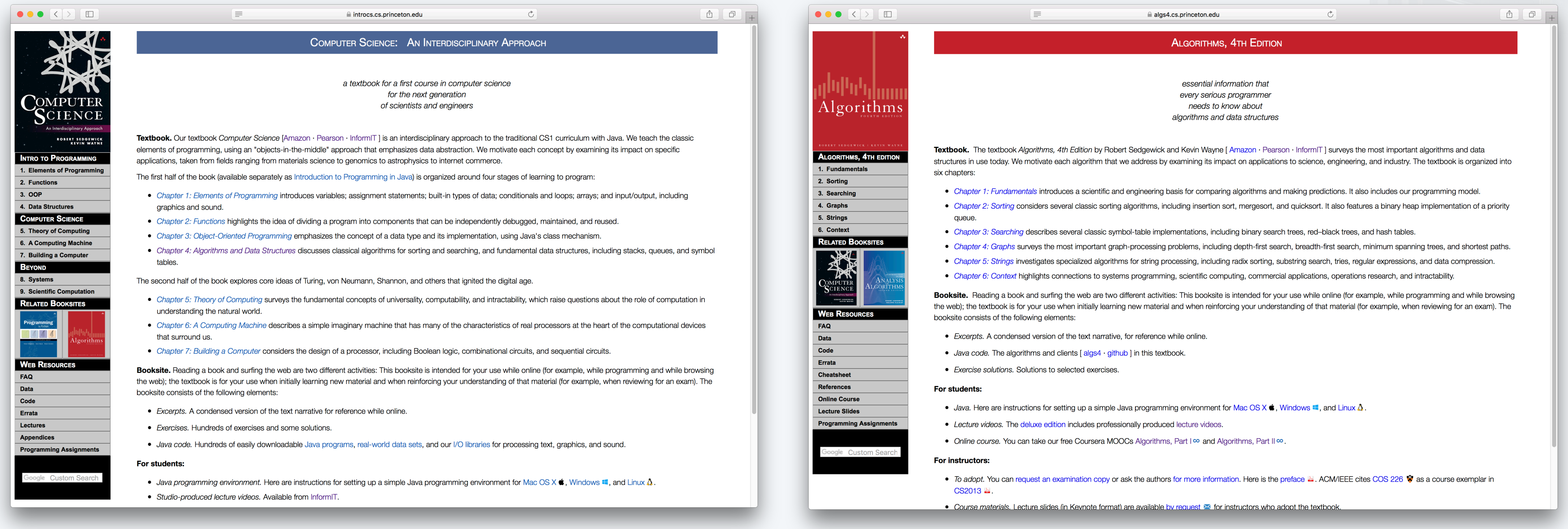
textbooks



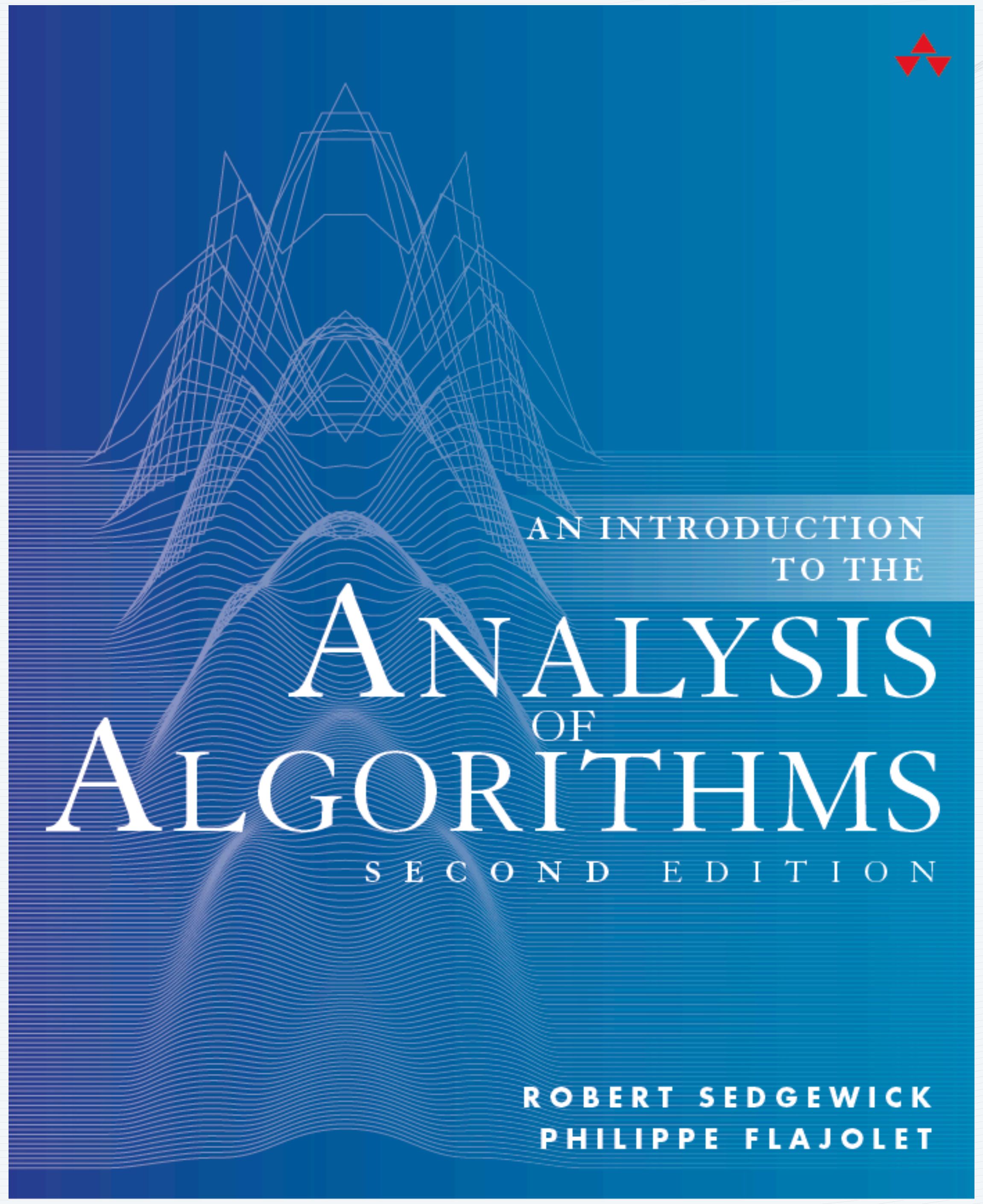
online lectures



booksites



... or whatever other resources you might have used to learn these topics



http://aofa.cs.princeton.edu

Cardinality Estimation

Robert Sedgewick
Princeton University

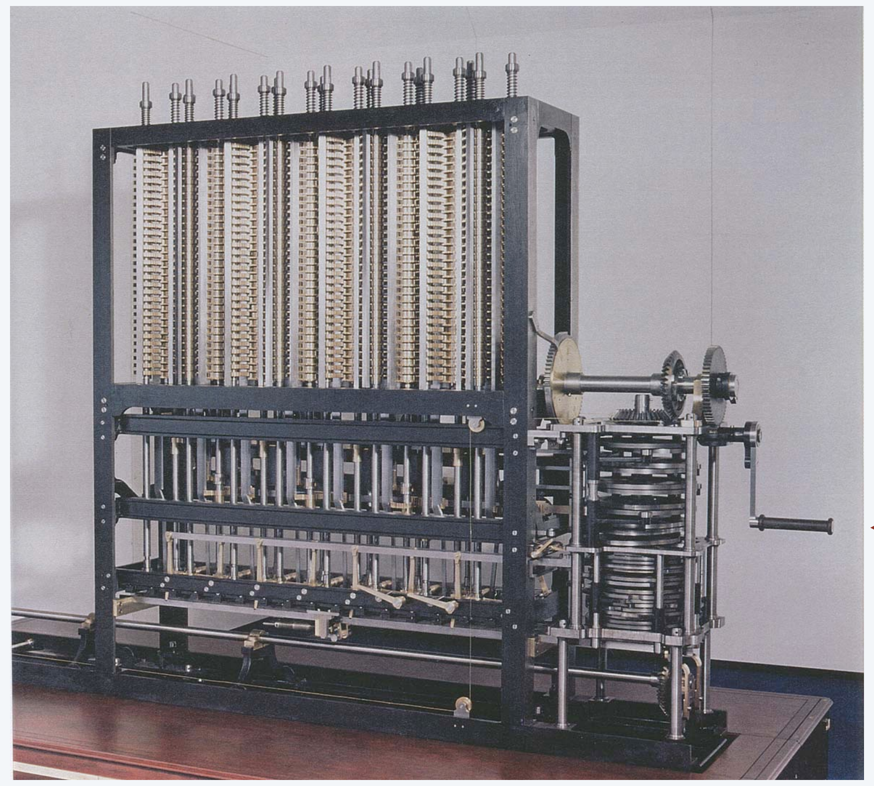
with special thanks to Jérémie Lumbroso

Cardinality Estimation

- Exact cardinality count
- Probabilistic counting
- Stochastic averaging
- Refinements
- Final frontier

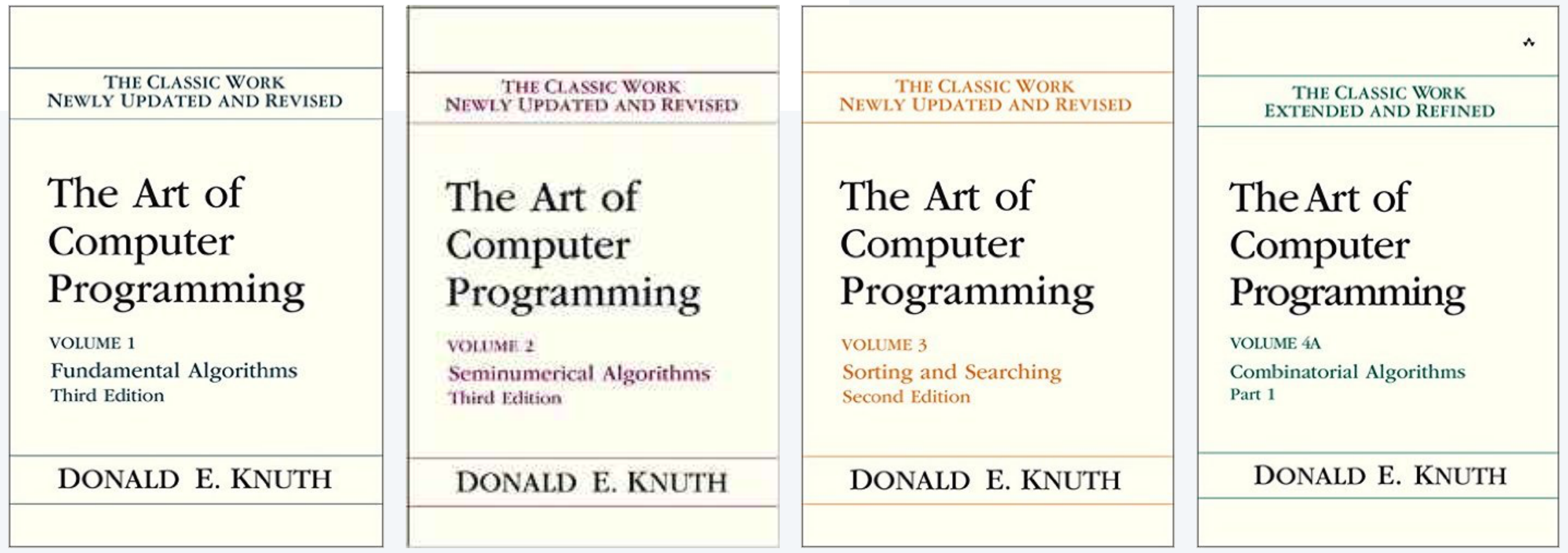
Don Knuth's legacy: Analysis of Algorithms (AofA)

Understood since Babbage:
- Computational resources are limited.
- Method (algorithm) used matters.


Analytic Engine

how many times do we have to turn the crank?

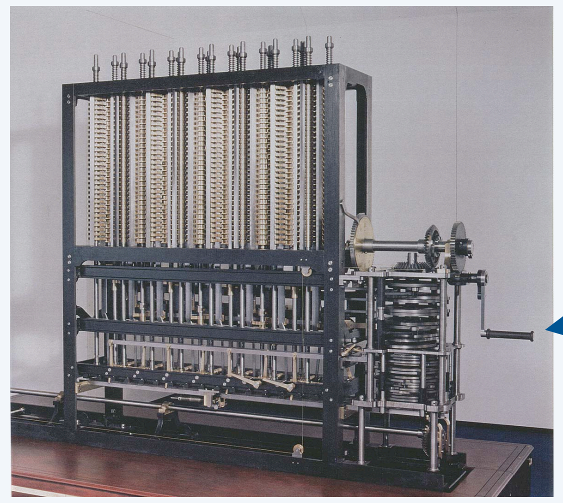
Knuth's insight: AofA is a *scientific* endeavor.
- Start with a working program (algorithm implementation).
- Develop mathematical model of its behavior.
- Use the *model* to formulate hypotheses on resource usage.
- Use the *program* to validate hypotheses.
- Iterate on basis of insights gained.



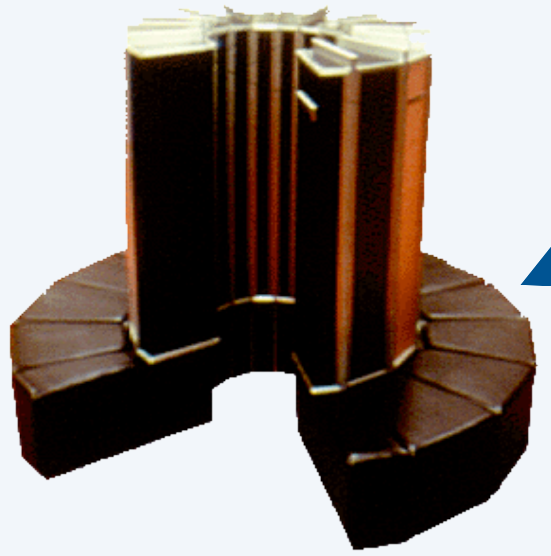
Difficult to overstate the significance of this insight.

AofA has played a critical role

in the development of our computational infrastructure *and the advance of scientific knowledge*



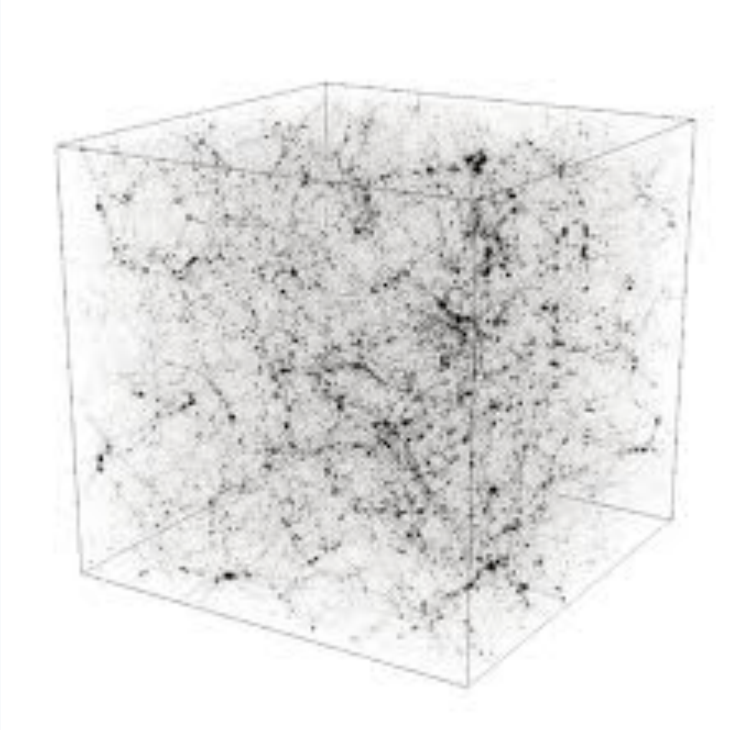
how many times
to turn the crank?



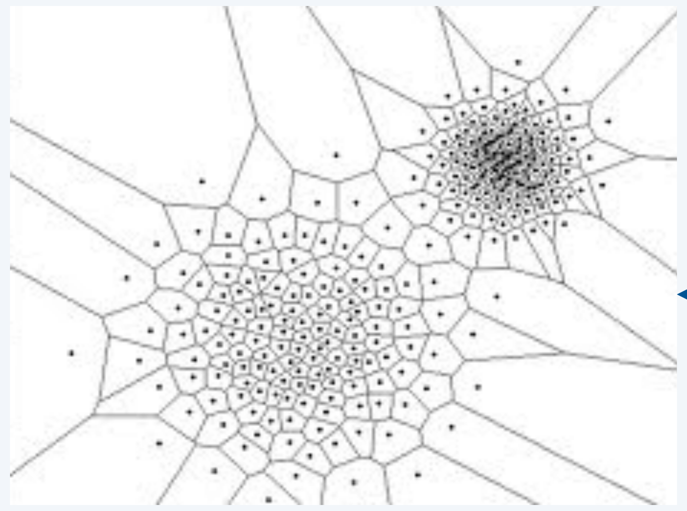
how long to sort random data for
cryptanalysis preprocessing?

how long to compile
my program?

how long to check
that my VLSI circuit
follows the rules?



how many bodies
in motion can I
simulate?

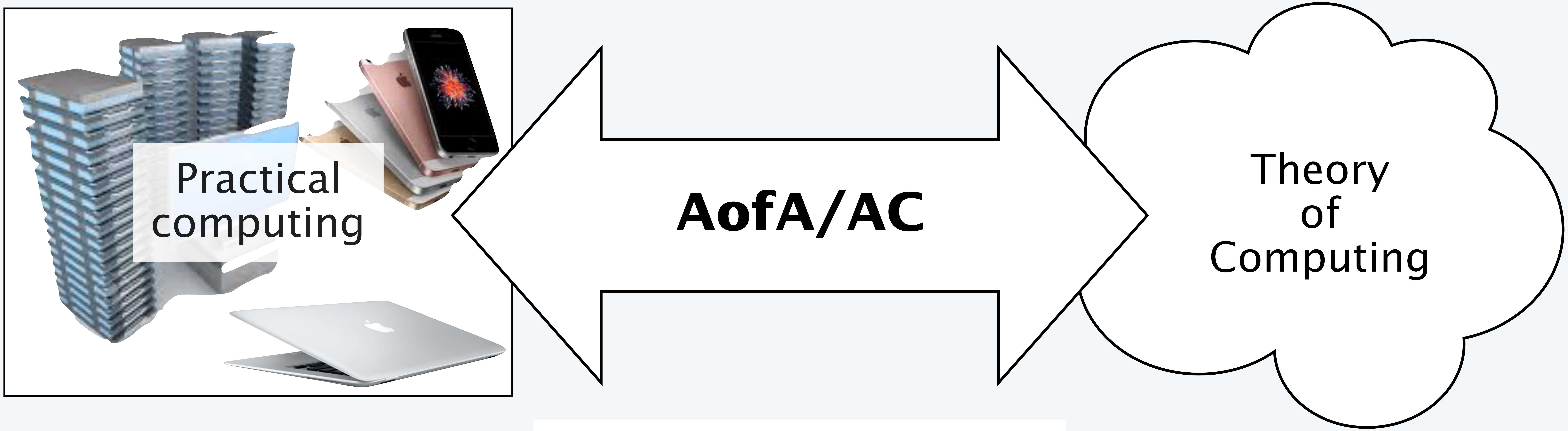


how quickly can I find clusters?

"PEOPLE WHO ANALYZE ALGORITHMS have double happiness. They experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically."

– Don Knuth

AofA/AC context



Practical computing

- Real code on real machines
- Thorough validation
- Limited math models

AofA/AC

- Theorems *and* code
- Scientific approach
- Experiment, validate, iterate

Theory of computing

- Theorems
- Abstract math models
- Limited experimentation

A case in point: Cardinality counting

Q. In a given stream of data values, how many different values are present?

Reference application. How many unique visitors in a web log?

log.07.f3.txt

```
109.108.229.102
pool-71-104-94-246.lsanca.dsl-w.verizon.net
117.222.48.163
pool-71-104-94-246.lsanca.dsl-w.verizon.net
1.23.193.58
188.134.45.71
1.23.193.58
gsearch.CS.Princeton.EDU
pool-71-104-94-246.lsanca.dsl-w.verizon.net
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
CPE-121-218-151-176.lnse3.cht.bigpond.net.au
```

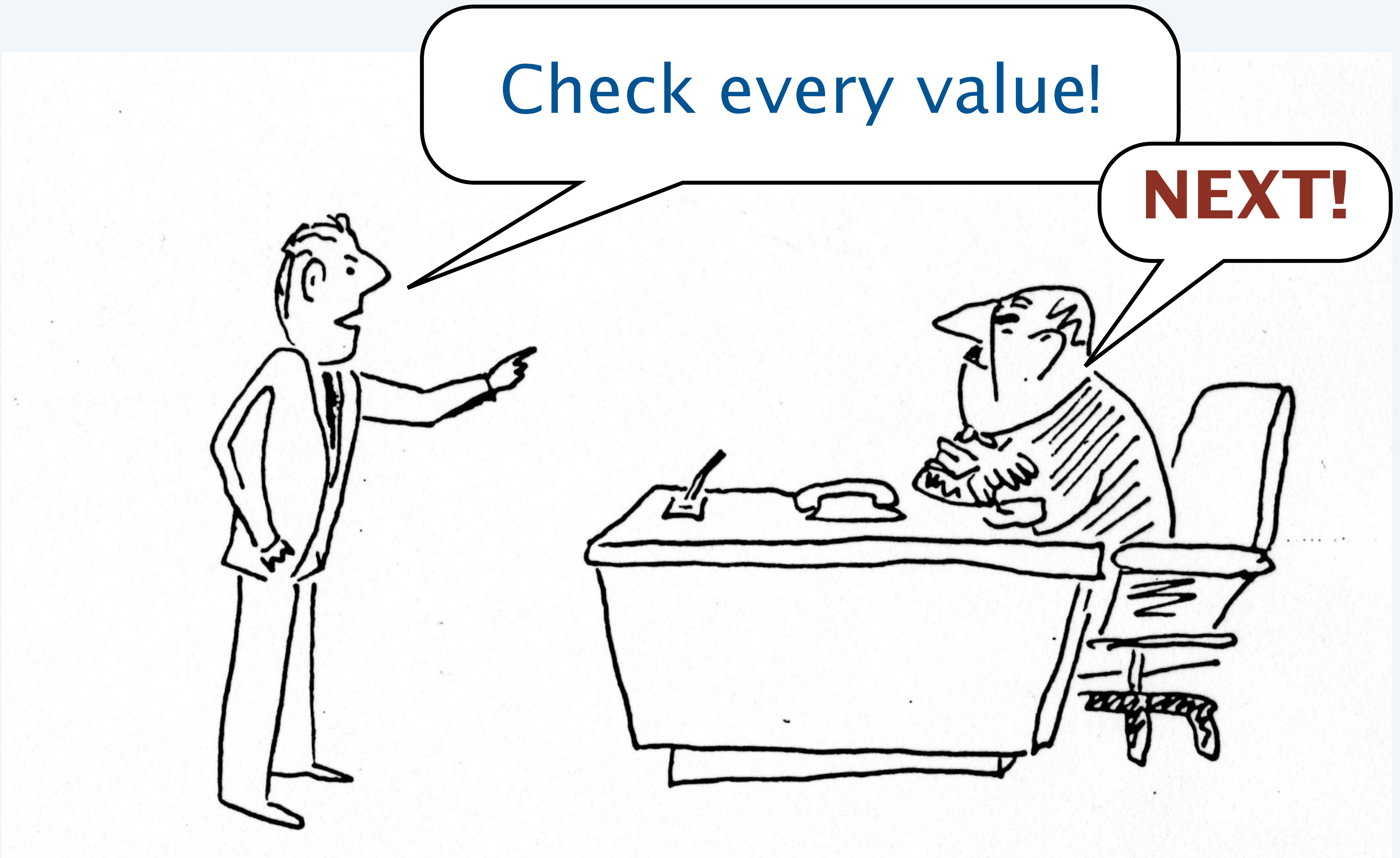
6 million strings

A standard "Interview Question".

Wrong answer: Check every value

Check every value
- Save all the values in an array.
- Check all previous values for duplicates.
- Count a value only if no previous duplicate.

```
int[] a = StdIn.readAllLines();
int count = 1;
for (int i = 1; i < a.length; i++)
{
   for (int j = 0; j <= i)
      if (a[j] == a[i]) break;
   if (j != i) count++;
}
StdOut.print(count + " different values");
```



Q. Why is this the wrong answer?

A. QUADRATIC running time, *therefore not feasible for real-world applications.*

Standard answer I: Sort, then count

Sort, then count

- Save all the values in an array.
- Sort the array.
- Equal values are together in the sorted input.
- Count the first occurrence of each value.

small example

15	9	9	4	10	9	11	12	10	14	12	11	15	6	11	9	8	5	10	2

sorted

2	4	5	6	8	9	9	9	9	10	10	10	11	11	11	11	12	14	15	15

count

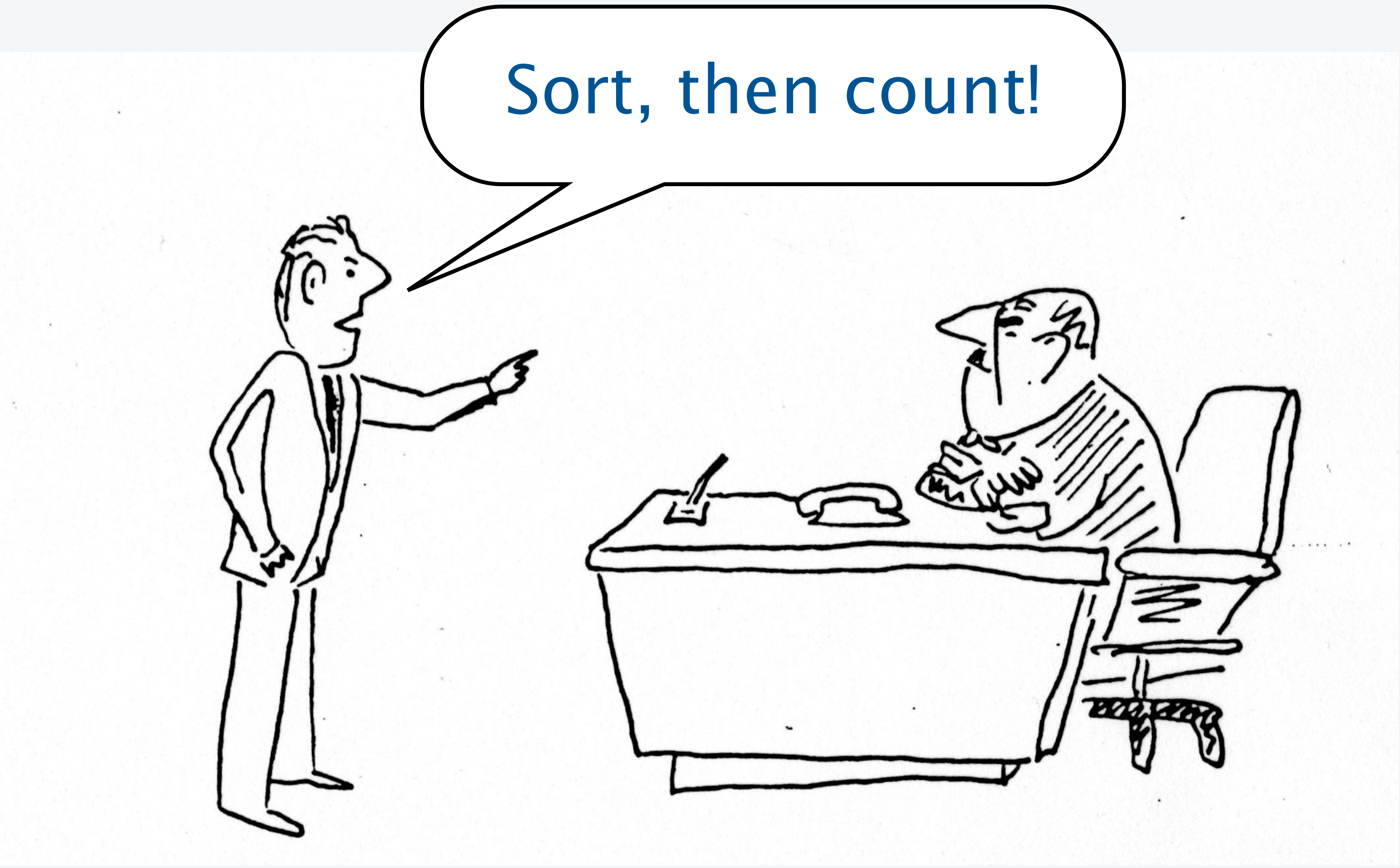
1	2	3	4	5	6				7			8				9	10	11	

increment counter when current value differs from previous value

Standard answer I: Sort, then count

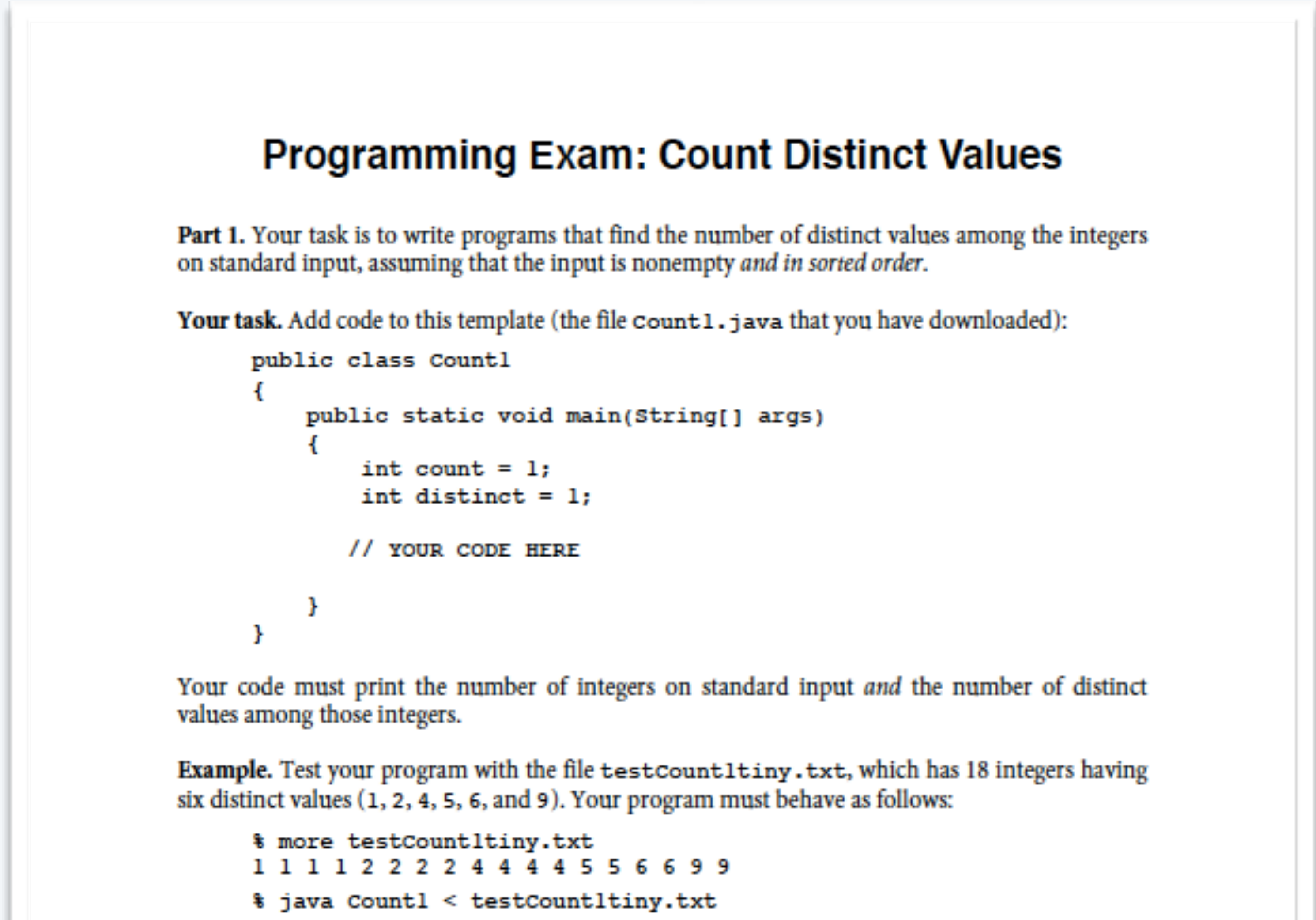
Sort, then count

- Save all the values in an array.
- Sort the array.
- Equal values are together in the sorted input.
- Count the first occurrence of each value.




```
int[] a = StdIn.readAllLines();
Arrays.sort(a);
int distinct = 1;
for (int i = 1; i < a.length; i++)
    if (a[i] != a[i-1]) distinct++;
StdOut.print(distinct + " different values");
```

Programming Exam 1 COS 126 2015



Programming Exam: Count Distinct Values

Part 1. Your task is to write programs that find the number of distinct values among the integers on standard input, assuming that the input is nonempty *and in sorted order.*

Your task. Add code to this template (the file `Count1.java` that you have downloaded):

```
public class Count1
{
    public static void main(String[] args)
    {
        int count = 1;
        int distinct = 1;

       // YOUR CODE HERE

    }
}
```

Your code must print the number of integers on standard input *and* the number of distinct values among those integers.

Example. Test your program with the file `testCount1tiny.txt`, which has 18 integers having six distinct values (`1`, `2`, `4`, `5`, `6`, and `9`). Your program must behave as follows:

```
% more testCount1tiny.txt
1 1 1 1 2 2 2 2 4 4 4 4 5 5 6 6 9 9
% java Count1 < testCount1tiny.txt
```

Used by programmers "in the wild" for decades

UNIX (1970s-present)

```
% sort -u log.07.f3.txt | wc -l
1112365
```

"unique"

Aside: Existence table

IF the values are positive integers less than U:

Use an *existence table*
- Create an array `b[]` of `boolean` values.
- For value `i`, set `b[i]` to `true`.
- Count the number of `true` values in `b[]`.

small example

15	9	9	4	10	9	11	12	10	14	12	11	15	6	11	9	8	5	10	2

existence table (U = 16)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		T		**T**	**T**	**T**		**T**	**T**	**T**	**T**	**T**		**T**	**T**

✘ Not applicable to reference application (long strings) because U would be prohibitively large.

Standard answer II: Use a hash table

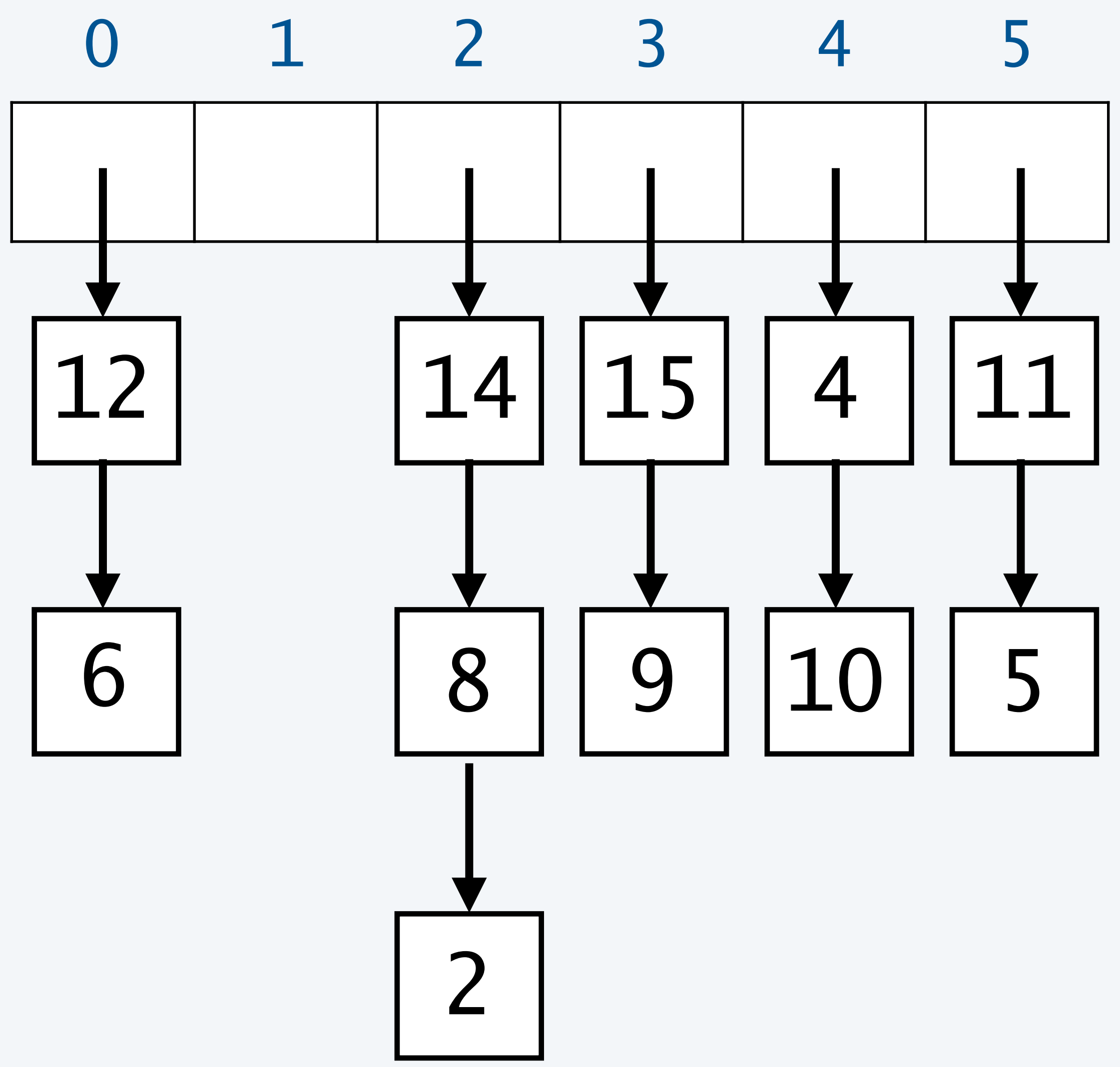
Hashing with separate chaining

- Create a table of size *M*.
- Transform each value into a “random” table index.
- Make linked lists for colliding values.
- Ignore values already in the table.

example: multiply by a prime,
then take remainder after dividing by *M*.

small example data stream	15	9	9	4	10	9	11	12	10	14	12	11	15	6	11	9	8	5	10	2
hash values (x*97 % 6)	3	3	3	4	4	3	5	0	4	2	0	5	3	0	5	3	2	5	4	2

hash table (M = 6)



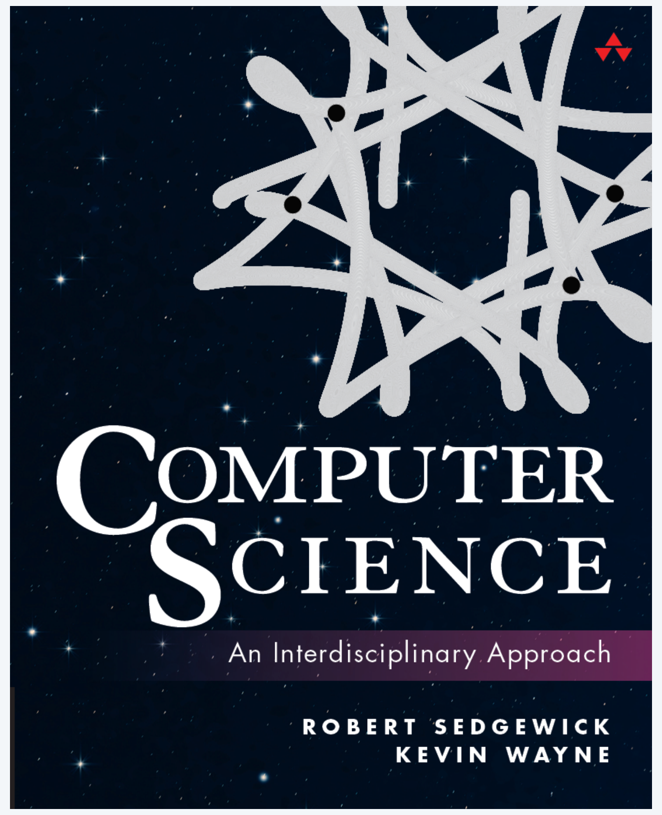
KEY IDEA. Keep lists short by resizing table.

Exact cardinality count using a hash table

Hashing with separate chaining

- Create a table of size *M*.
- Transform each value into a "random" table index.
- Make linked lists for colliding values.
- Ignore values already in the table.

Widely used and well studied textbook method.



Exact cardinality count in Java

- Input is an "iterable"
- HashSet implements a hash table
- add() adds new value (noop if already there)
- size() gives number of distinct values added

```
public static long count(Iterable<String> stream)
{
   HashSet<String> hset = new HashSet<String>();
   for (String x : stream)
       hset.add(x);
   return hset.size();
}
```

Mathematical analysis of exact cardinality count with hashing

Theorem. *If the hash function uniformly and independently distributes the keys in the table*, the expected time and space cost is LINEAR.

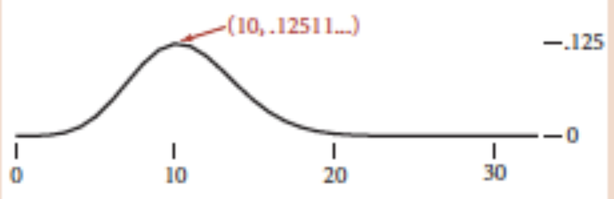
Proof. See Proposition K
in *Algorithms*, page 466.

based on classic probability theory
(binomial and Poisson distributions)

Proposition K. In a separate-chaining hash table with M lists and N keys, the probability (under ASSUMPTION J) that the number of keys in a list is within a small constant factor of N/M is extremely close to 1.

Proof sketch: ASSUMPTION J makes this an application of classical probability theory. We sketch the proof, for readers who are familiar with basic probabilistic analysis. The probability that a given list will contain exactly k keys is given by the *binomial distribution*

$$\binom{N}{k}\left(\frac{1}{M}\right)^k\left(\frac{M-1}{M}\right)^{N-k}$$



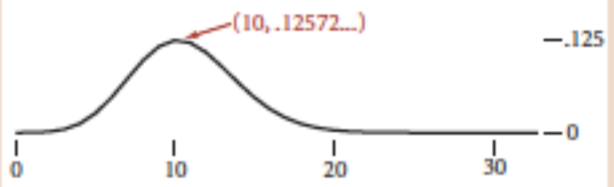
Binomial distribution ($N = 10^4$, $M = 10^3$, $\alpha = 10$)

by the following argument: Choose k out of the N keys. Those k keys hash to the given list with probability $1/M$, and the other $N - k$ keys do not hash to the given list with probability $1 - (1/M)$. In terms of $\alpha = N/M$, we can rewrite this expression as

$$\binom{N}{k}\left(\frac{\alpha}{N}\right)^k\left(1-\frac{\alpha}{N}\right)^{N-k}$$

which (for small α) is closely approximated by the classical *Poisson distribution*

$$\frac{\alpha^k e^{-\alpha}}{k!}$$



Poisson distribution ($N = 10^4$, $M = 10^3$, $\alpha = 10$)

It follows that the probability that a list has more than $t\,\alpha$ keys on it is bounded by the quantity $(\alpha\, e/t)^t e^{-\alpha}$. This probability is extremely small for practical ranges of the parameters. For example, if the average length of the lists is 10, the probability that we will hash to some list with more than 20 keys on it is less than $(10\, e/2)^2 e^{-10} \approx 0.0084$, and if the average length of the lists is 20, the probability that we will hash to some list with more than 40 keys on it is less than $(20\, e/2)^2 e^{-20} \approx 0.0000016$. This concentration result does not guarantee that *every* list will be short. Indeed it is known that, if α is a constant, the average length of the longest list grows with $\log N / \log \log N$.

Q. Do the hash functions that we use *uniformly and independently distribute keys in the table*?

A. Not likely.

Scientific validation of exact cardinality count with linear probing

Hypothesis. Time and space cost is *linear* *for the hash functions we use and the data we have.*

Quick experiment. Doubling the problem size should double the running time.

Driver to read N strings and count distinct values

get problem size
initialize input stream
get current time
print count
print elapsed time

```
public static void main(String[] args)
{
   int N = Integer.parseInt(args[0]);
   StringStream stream = new StringStream(N);
   long start = System.currentTimeMillis();

   StdOut.println(count(stream));

   long now = System.currentTimeMillis();
   double time = (now - start) / 1000.0;
   StdOut.println(time + " seconds");
}
```

```
% java Hash 2000000 < log.07.f3.txt
483477
3.322 seconds

% java Hash 4000000 < log.07.f3.txt
883071
6.55 seconds

% java Hash 6000000 < log.07.f3.txt
1097944
9.49 seconds
```

✓

```
% sort -u log.07.f3 | wc -l
1097944
```

sort-based method takes about 3 minutes

Q. Is hashing with linear probing effective?

A. Yes! Validated in countless applications for *over half a century*.

Summary of cardinality count algorithms

	time bound	*memory bound*
Wrong answer	N^2	N
Sort and count	$N \log N$	N
Existence table	N	U
Hash table	N *	N

* Theoretical AofA. Hashing solution is *quadratic* in the worst case.

Theoretical AofA. If (uniform hashing assumption) then *hashing solution is linear* (expected).

Scientific AofA. Hypothesis that *hashing solution is linear* has been validated for decades.

Q. End of story?

A. No. *Beginning* of story!

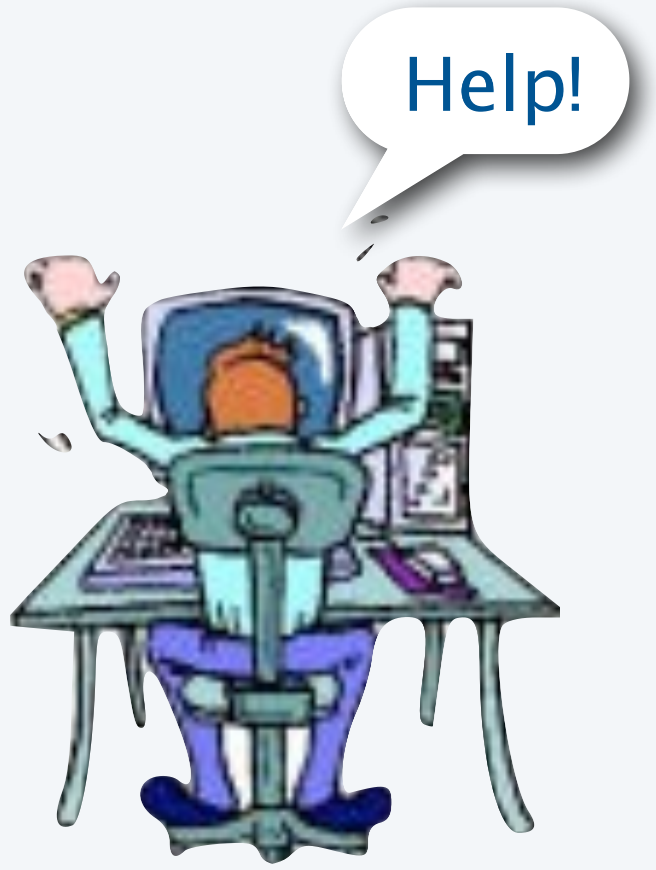
A problem: Exact cardinality count requires linear space

Q. I can't use a hash table. The stream is much too big to fit all values in memory. Now what?

A. Bad news: You cannot get an exact count.

A. (Bloom, 1970) You can get an accurate *estimate* using a few bits per distinct value.

```
109.108.229.102
pool-71-104-94-246.lsanca.dsl-w.verizon.net
117.222.48.163
pool-71-104-94-246.lsanca.dsl-w.verizon.net
1.23.193.58
188.134.45.71
1.23.193.58
gsearch.CS.Princeton.EDU
pool-71-104-94-246.lsanca.dsl-w.verizon.net
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
CPE-121-218-151-176.lnse3.cht.bigpond.net.au
117.211.88.36
msnbot-131-253-46-251.search.msn.com
msnbot-131-253-46-251.search.msn.com
```



A. Much better news: *You can get an accurate estimate using only a handful of bits* (stay tuned).

Cardinality Estimation

- Warmup: exact cardinality count
- **Probabilistic counting**
- Stochastic averaging
- Refinements
- Final frontier

Cardinality *estimation*

is a fundamental problem with many applications *where memory is limited*.

Q. *About* how many different values appear in a given stream?

Constraints

- Make *one pass* through the stream.
- Use *as few operations per value* as possible
- Use *as little memory* as possible.
- Produce *as accurate an estimate* as possible.

typical applications

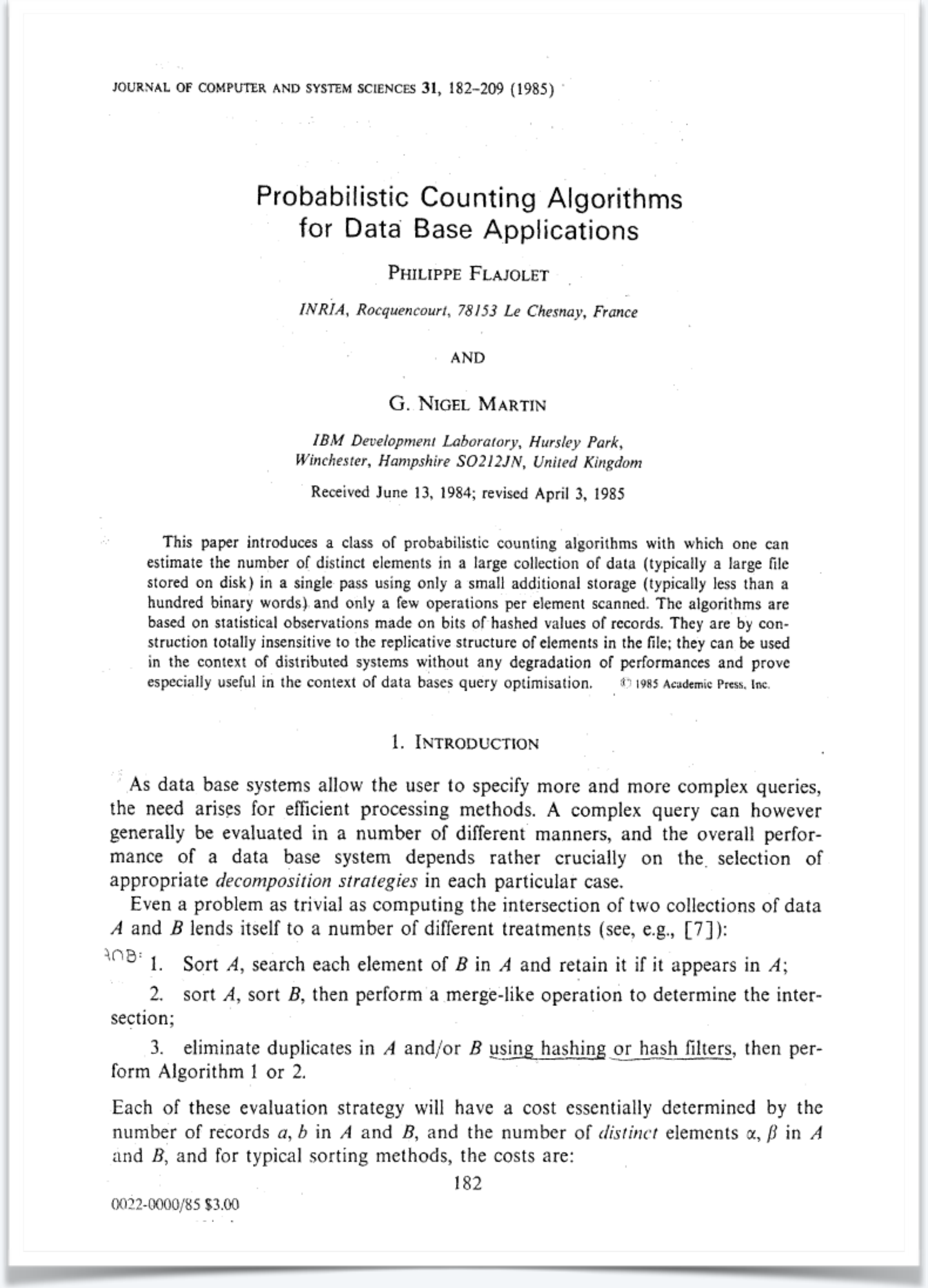
- How many unique visitors to my website?
- Which sites are the most/least popular?
- How many different websites visited by each customer?
- How many different values for a database join?

To fix ideas on scope: Think of *billions* of streams each having *trillions* of values.

Probabilistic counting with stochastic averaging (PCSA)

Flajolet and Martin, *Probabilistic Counting Algorithms for Data Base Applications FOCS* 1983, *JCSS* 1985.

Philippe Flajolet 1948–2011



JOURNAL OF COMPUTER AND SYSTEM SCIENCES **31**, 182–209 (1985)

Probabilistic Counting Algorithms for Data Base Applications

PHILIPPE FLAJOLET

INRIA, Rocquencourt, 78153 Le Chesnay, France

AND

G. NIGEL MARTIN

IBM Development Laboratory, Hursley Park, Winchester, Hampshire SO212JN, United Kingdom

Received June 13, 1984; revised April 3, 1985

This paper introduces a class of probabilistic counting algorithms with which one can estimate the number of distinct elements in a large collection of data (typically a large file stored on disk) in a single pass using only a small additional storage (typically less than a hundred binary words) and only a few operations per element scanned. The algorithms are based on statistical observations made on bits of hashed values of records. They are by construction totally insensitive to the replicative structure of elements in the file; they can be used in the context of distributed systems without any degradation of performances and prove especially useful in the context of data bases query optimisation.

1. INTRODUCTION

As data base systems allow the user to specify more and more complex queries, the need arises for efficient processing methods. A complex query can however generally be evaluated in a number of different manners, and the overall performance of a data base system depends rather crucially on the selection of appropriate *decomposition strategies* in each particular case.

Even a problem as trivial as computing the intersection of two collections of data A and B lends itself to a number of different treatments (see, e.g., [7]):

1. Sort A, search each element of B in A and retain it if it appears in A;
2. sort A, sort B, then perform a merge-like operation to determine the intersection;
3. eliminate duplicates in A and/or B using hashing or hash filters, then perform Algorithm 1 or 2.

Each of these evaluation strategy will have a cost essentially determined by the number of records a, b in A and B, and the number of *distinct* elements α, β in A and B, and for typical sorting methods, the costs are:

182

0022-0000/85 $3.00

Contributions

- Introduced problem
- Idea of *streaming algorithm*
- Idea of "small" *sketch* of "big" data
- Detailed analysis that yields tight bounds on accuracy
- Full validation of mathematical results with experimentation
- Practical algorithm that has remained effective for decades

Bottom line. Quintessential example of the effectiveness of scientific approach to algorithm design.

PCSA first step: Use hashing

Transform value to a "random" computer word.

- Compute a *hash function* that transforms data value into a 32- or 64-bit value.
- Cardinality count is unaffected (with high probability).
- Built-in capability in modern systems.
- *Allows use of fast machine-code operations.*

20th century: use 32 bits (millions of values)
21st century: use 64 bits (quadrillions of values)

Example: Java

- All data types implement a hashCode() method (though we often override the default).
- String data type stores value (computed once).

```
String value = "gsearch.CS.Princeton.EDU"
int x = value.hashCode();
```

current Java default is 32-bit int value

Bottom line: Do cardinality estimation on streams of (binary) integers.

```
01111000100111110111000111001000
01111000100111110111000111001000
01110101010110110000000011011010
00110100010001111100010100111010
00010000111001101000111010010011
00001001011011100000010010010111
00001001011011100000010010010111
```

"Random" *except* for the fact that some values are equal.

Initial hypothesis

Hypothesis. Uniform hashing assumption is reasonable in this context.

Implication. Need to run experiments to validate any hypotheses about performance.

No problem!

- AofA is a scientific endeavor (we always validate hypotheses).
- End goal is development of algorithms that are useful in practice.
- It is the responsibility of the *designer* to validate utility before claiming it.
- After decades of experience, discovering a performance problem due to a bad hash function would be a significant research result.

Unspoken bedrock principle of AofA.

Experimenting to validate hypotheses is **WHAT WE DO!**

Probabilistic counting starting point: two integer functions

Definition. $r(x)$ is the **number of *trailing* 1s** in the binary representation of x. ← *position of rightmost 0*

Definition. $R(x) = 2^{r(x)}$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	$r(x)$	$R(x)$	$R(x)_2$
1	0	1	1	1	1	0	1	1	1	1	1	0	1	0	1	**1**	**2**	**10**
1	0	1	0	1	0	1	0	1	0	0	0	1	1	1	0	**0**	**1**	**1**
0	1	1	0	1	0	0	1	0	1	0	**1**	**1**	**1**	**1**	**1**	**5**	**32**	**100000**

Bit-whacking basics:
$R(x)$ is easy to compute.

0	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	x
1	0	0	1	0	1	1	0	1	0	1	0	0	0	0	0	$\sim x$
0	1	1	0	1	0	0	1	0	1	1	0	0	0	0	0	$x + 1$
0	0	0	0	0	0	0	0	0	0	**1**	**0**	**0**	**0**	**0**	**0**	$\sim x \ \& \ (x + 1)$

3 instructions on a typical computer

Bit-whacking magic:
$r(x)$ is also "easy" to compute (don't ask).

see Knuth volume 4A., page 141

available as a single instruction on modern processors

Bottom line: $r(x)$ and $R(x)$ can be computed with *just a few machine instructions*.

Probabilistic counting (Flajolet and Martin, 1983)

Maintain a single-word *sketch* that summarizes a data stream $x_0, x_1, \ldots, x_N, \ldots$

- For each x_N in the stream, update sketch by *bitwise or* with $R(x_N)$.
- Use position of rightmost 0 in sketch to estimate $\lg N$.

estimate of lg N

typical sketch
***N* = 10⁶**

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sketch	0	0	0	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
x_N	0	0	1	1	0	1	0	1	0	1	1	1	1	1	1	0	1	0	1	0	1	0	1	0	1	0	0	0	1	1	1	1
$R(x_N)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
sketch | $R(x_N)$	0	0	0	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

$R(x) = 2^k$ with probability $1/2^k$

leading bits almost surely 0

trailing bits almost surely 1

Rough estimate of $\lg N$ is $r(sketch)$.

Rough estimate of N is $R(sketch)$.

correction factor needed (stay tuned)

Probabilistic counting trace

x	*r(x)*	*R(x)*	*sketch*
01100010011000111010011110111011	2	100	00000000000000000000000000000100
01100111001000110001111100000101	1	10	00000000000000000000000000000110
00010001000111000110110110110011	2	100	00000000000000000000000000000110
01000100011101110000000111011111	5	100000	00000000000000000000000000100110
01101000001011000101110001000100	0	1	00000000000000000000000000100111
00110111101100000001010010101011	1	10	00000000000000000000000000100111
00110100011000111010101111111100	0	1	00000000000000000000000000100111
00011000010000100001011100110111	3	1000	00000000000000000000000000101111
00011001100110011110010000111111	6	1000000	00000000000000000000000001101111
01000101110001001010110011111100	0	1	00000000000000000000000001101111

$R(sketch) = 10000_2$
$= 16$

Probabilistic counting (Flajolet and Martin, 1983)

```
public long R(long x)
{ return ~x & (x+1); }

public long estimate(Iterable<String> stream)
{
   long sketch;
   for (s : stream)
      sketch = sketch | R(s.hashCode());
   return R(sketch) /.77351;
}
```

Maintain a *sketch* of the data

- A single word
- OR of all values of R(x) in the stream
- Return smallest value not seen

  with correction for bias

Early example of "a simple algorithm whose analysis isn't"

Q. (Martin) Estimate seems a bit low. How much?

A. (unsatisfying) Obtain correction factor empirically.

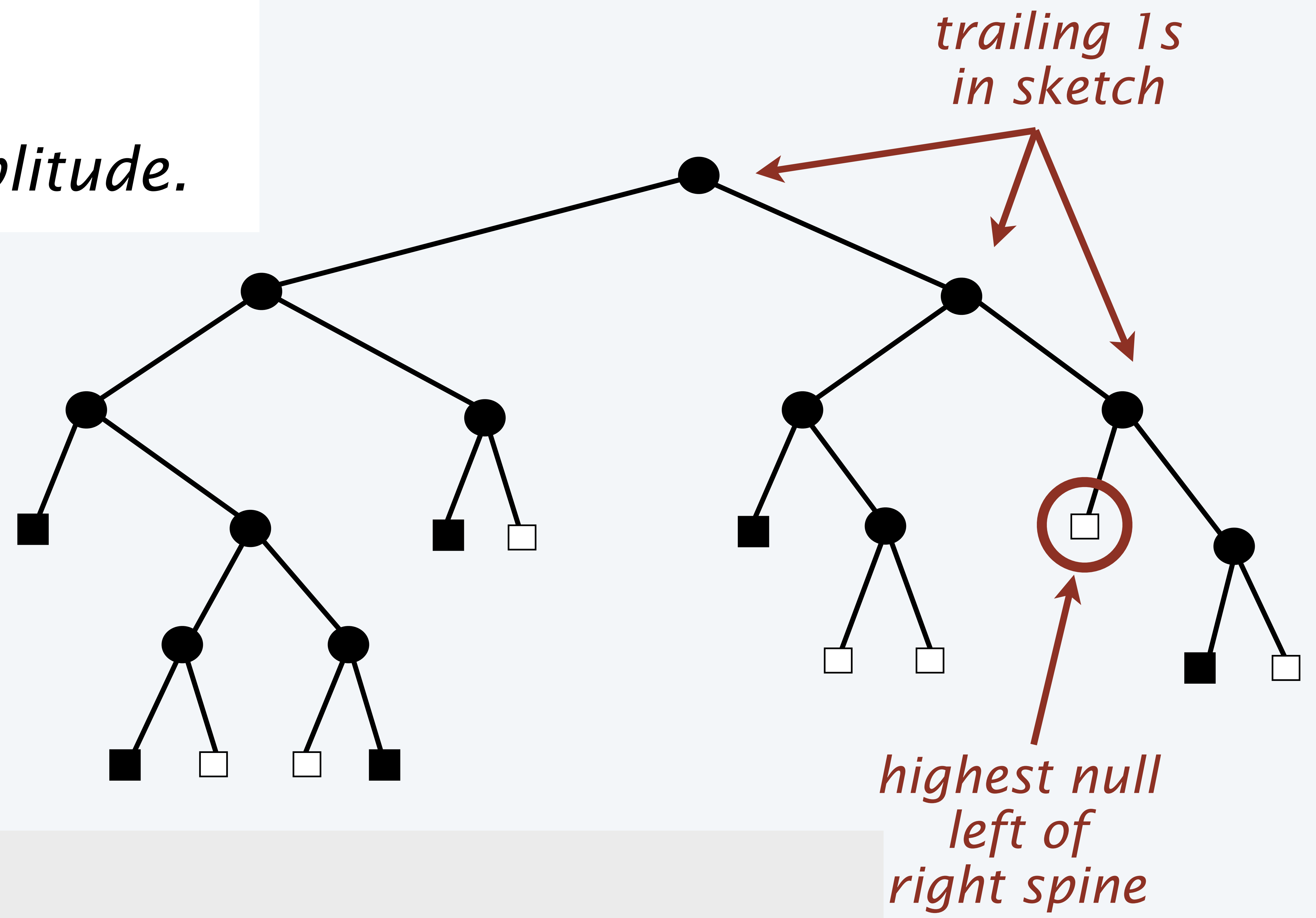
A. (Flajolet) Do the math. Without it, there is no algorithm!

Mathematical analysis of probabilistic counting

Theorem. *The expected number of trailing 1s in the PC sketch is*

$$\lg(\phi N) + P(\lg N) + o(1) \quad \textit{where} \ \ \phi \doteq .77351$$

and P is an oscillating function of lg *N of very small amplitude.*



Proof (omitted).

1980s: Flajolet *tour de force*

1990s: trie parameter

21st century: standard analytic combinatorics

Kirschenhofer, Prodinger, and Szpankowski
Analysis of a splitting process arising in probabilistic counting and other related algorithms, ICALP 1992.

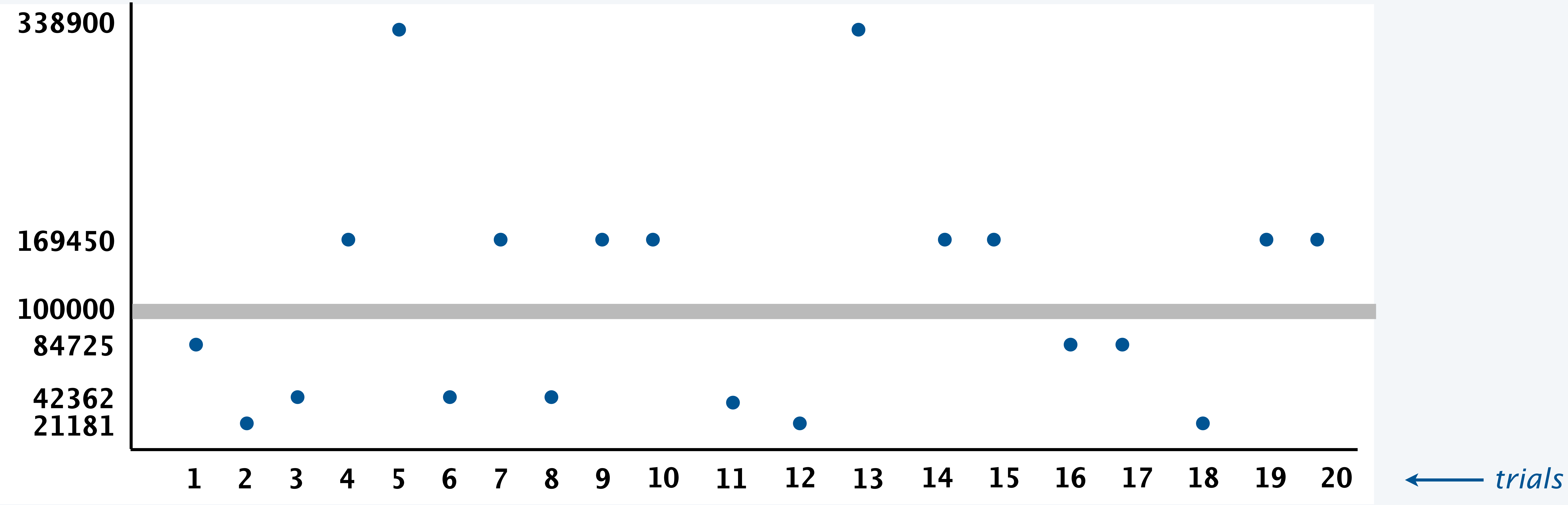
Jacquet and Szpankowski
Analytical depoissonization and its applications, TCS 1998.

In other words. In PC code, `R(sketch)/.77351` is an *unbiased statistical estimator* of *N*.

Validation of probabilistic counting

Hypothesis. Expected value returned is N *for random values from a large range.*

Experiment. 100,000 31-bit random values (20 trials)



Flajolet and Martin: Result is "typically one binary order of magnitude off."

Of course! (Always returns a power of 2 divided by .77351.)

Need to incorporate more experiments for more accuracy.

16384/.77351 = 21181
32768/.77351 = 42362
65536/.77351 = 84725
131072/.77351 = 169450
...

Cardinality Estimation

- Rules of the game
- Probabilistic counting
- **Stochastic averaging**
- Refinements
- Final frontier

Stochastic averaging

Goal. Perform M independent PC experiments and average results.

Alternative 1: M independent hash functions? *No, too expensive (and wasteful).*

Alternative 2: M-way alternation? *No, bad results for certain inputs.*

01 02 03 04 01 02 03 04 01 02 03 04

01 01 01
02 02 02
03 03 03
04 04 04

Alternative 3: *Stochastic averaging*

- Use second hash to divide stream into 2^m independent streams
- Use PC on each stream, yielding 2^m sketches .
- Compute *mean* = average number of trailing bits in the sketches.
- Return $2^{mean}/.77531$.

key point: equal values all go to the same stream

11 09 07 23 31 07 22 22 10 11 39 21

09 07 07
11 10 11
23 22 22 21
31 39

PCSA trace

$M = 4$

use initial m bits for second hash

x	*R(x)*	*sketch[0]*	*sketch[1]*	*sketch[2]*	*sketch[3]*
1010011110111011	100	0000000000000000	0000000000000000	0000000000000100	0000000000000000
0001111100000101	10	0000000000000010	0000000000000000	0000000000000100	0000000000000000
0110110110110011	100	0000000000000010	0000000000000100	0000000000000100	0000000000000000
0000000111011111	100000	0000000000100010	0000000000000100	0000000000000100	0000000000000000
0101110001000100	1	0000000000100010	0000000000000101	0000000000000100	0000000000000000
0000101001010101	10	0000000000100010	0000000000000101	0000000000000100	0000000000000000
1010101111111100	1	0000000000100010	0000000000000101	0000000000000101	0000000000000000
0001011100110111	1000	0000000000101010	0000000000000101	0000000000000101	0000000000000000
1110010000111111	1000000	0000000000101010	0000000000000101	0000000000000101	0000000001000000
1010110011111101	10	0000000000101010	0000000000000101	0000000000000111	0000000001000000
0001110100110100	1	0000000000101011	0000000000000101	0000000000000111	
		0000000000101011	0000000000000101	0000000000000111	0000000001000000
r (sketch[ ])		*2*	*1*	*3*	*0*

Probabilistic counting with stochastic averaging in Java

```
public static long estimate(Iterable<Long> stream, int M)
{
   long[] sketch = new long[M];
   for (long x : stream)
   {
      int k = hash2(x, M);
      sketch[k] = sketch[k] | R(x);
   }
   int sum = 0;
   for (int k = 0; k < M; k++)
      sum += r(sketch[k]);
   double mean = 1.0 * sum / M;
   return (int) (M * Math.pow(2, mean)/.77351);
}
```

Flajolet-Martin 1983

Idea. *Stochastic averaging*

- Use hash to convert stream to integers and compute R values as before
- Use second hash to split into $M = 2^m$ independent streams
- Use PC on each stream, yielding 2^m sketches .
- Compute *mean* = average # trailing 1 bits in the sketches.
- Return $2^{mean}/.77351$.

Observation. Accuracy improves as M increases.

Q. By how much?

Theorem (paraphrased to fit context of this talk).

Under the uniform hashing assumption, PCSA

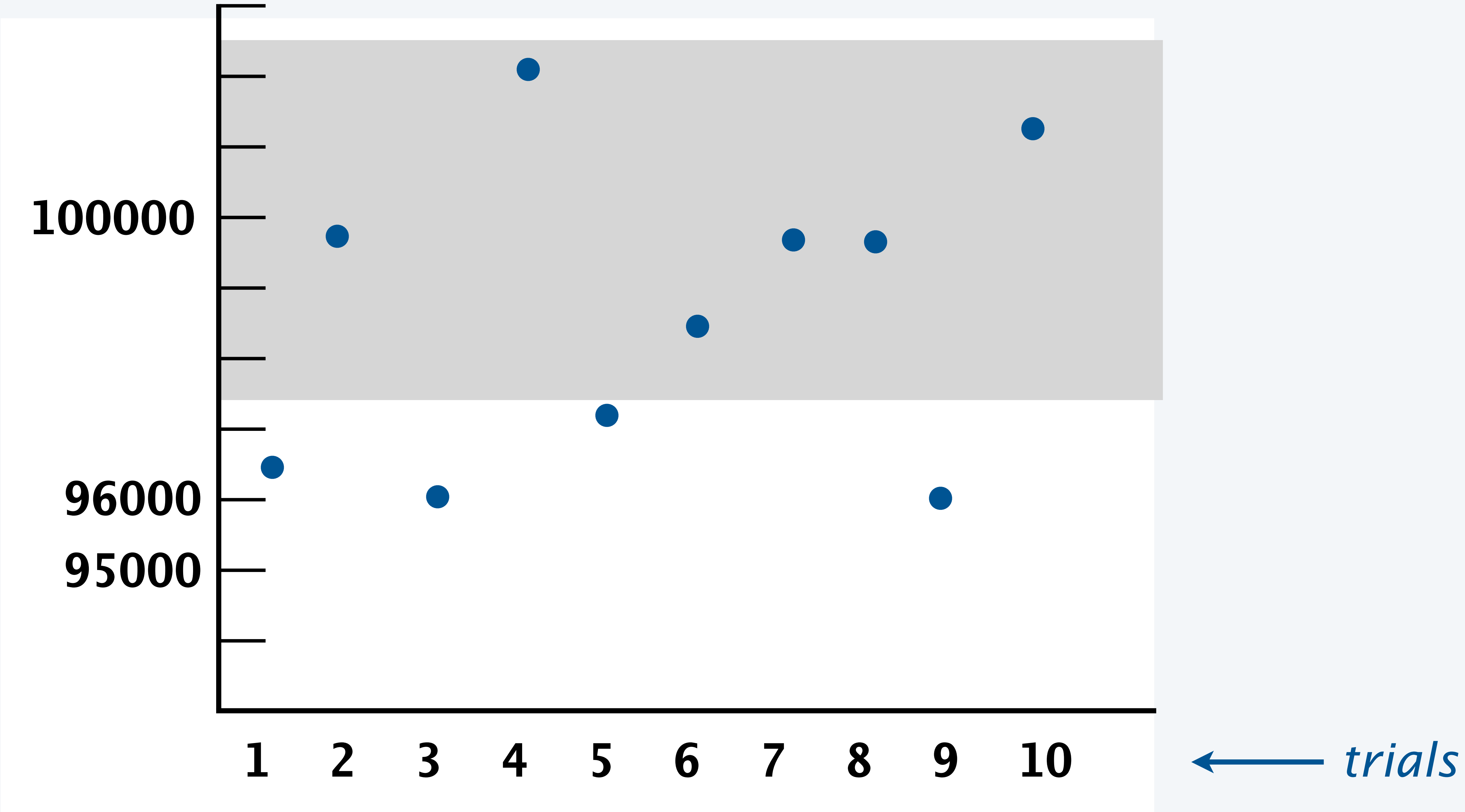
- *Uses 64M bits.*
- *Produces estimate with a relative accuracy close to* $0.78/\sqrt{M}$

Validation of PCSA analysis

Hypothesis. Value returned is accurate to $0.78/\sqrt{M}$ *for random values from a large range.*

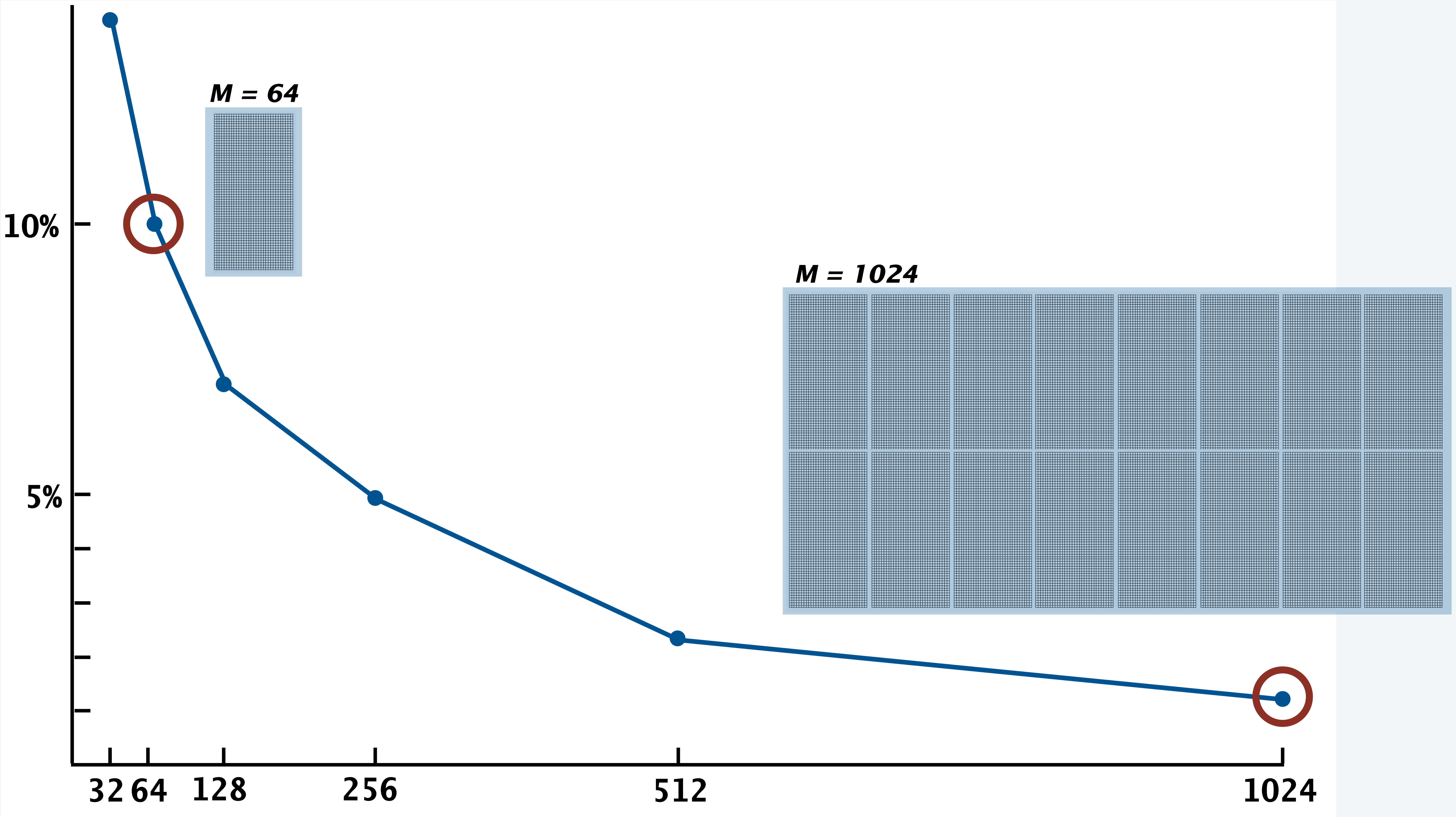
Experiment. 100,000 31-bit random values (10 trials)

```
% java PCSA 1000000 31 1024 10
964416
997616
959857
1024303
972940
985534
998291
996266
959208
1015329
```



Space-accuracy tradeoff for probabilistic counting with stochastic averaging

Relative accuracy: $\dfrac{0.78}{\sqrt{M}}$



Bottom line.

- Attain 10% relative accuracy with a sketch consisting of 64 words.
- Attain 2.4% relative accuracy with a sketch consisting of 1024 words.

Scientific validation of PCSA

Hypothesis. Accuracy is as specified *for the hash functions we use and the data we have.*

Validation (Flajolet and Martin, 1985). Extensive reproducible scientific experiments (!)

Validation (RS, this morning).

```
% java PCSA 6000000 1024 < log.07.f3.txt
1106474
```

<1% larger than actual value

log.07.f3.txt

```
109.108.229.102
pool-71-104-94-246.lsanca.dsl-w.verizon.net
117.222.48.163
pool-71-104-94-246.lsanca.dsl-w.verizon.net
1.23.193.58
188.134.45.71
1.23.193.58
gsearch.CS.Princeton.EDU
pool-71-104-94-246.lsanca.dsl-w.verizon.net
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
81.95.186.98.freenet.com.ua
CPE-121-218-151-176.lnse3.cht.bigpond.net.au
```

Q. Is PCSA effective?

A. ABSOLUTELY!

Summary: PCSA (Flajolet-Martin, 1983)

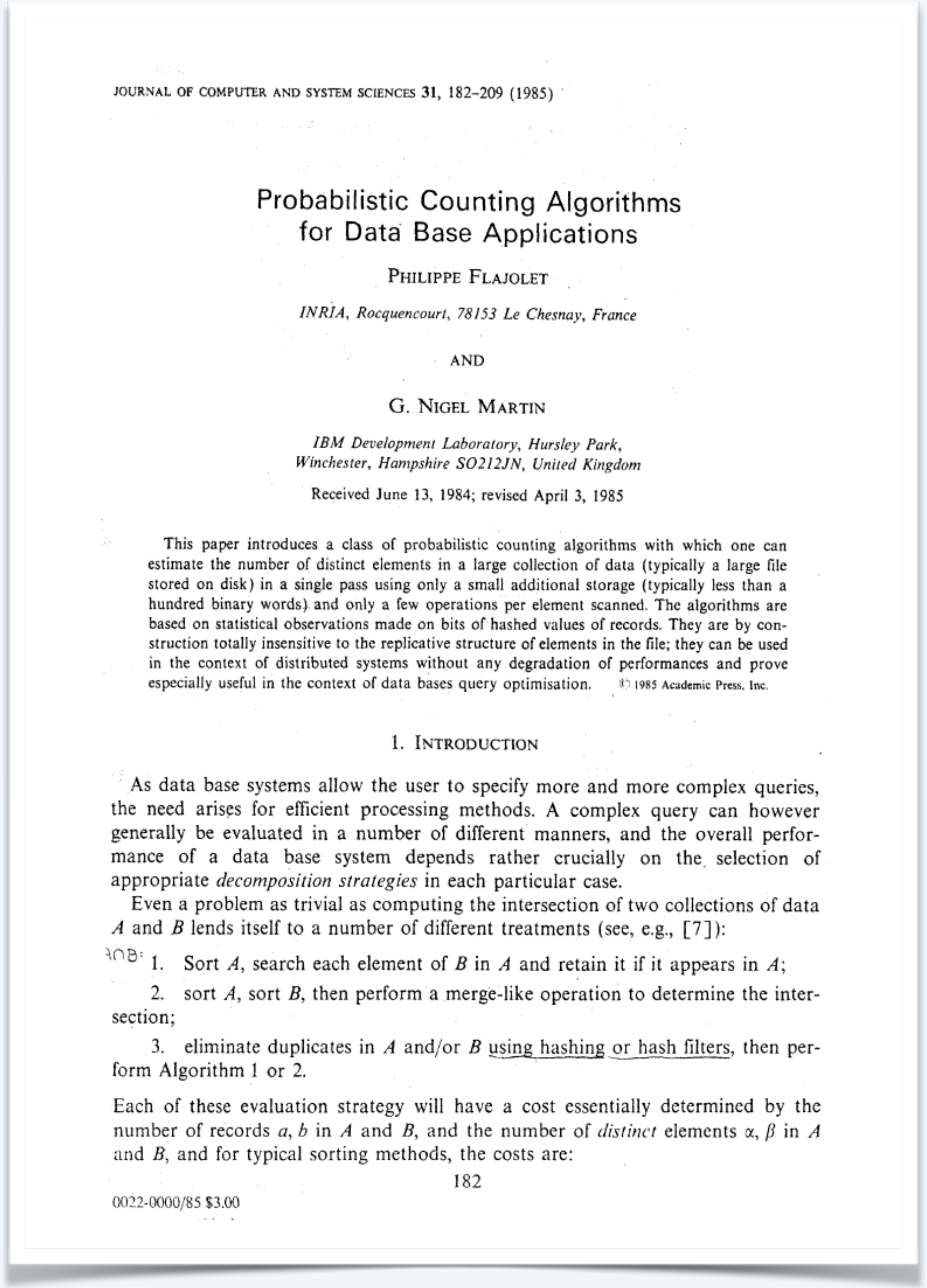
is a *demonstrably* effective approach to cardinality estimation

Q. *About* how many different values are present in a given stream?

PCSA

- Makes *one pass* through the stream.
- Uses *a few machine instructions per value*
- Uses *M* words to achieve relative accuracy $0.78/\sqrt{M}$ ✓

Results validated through extensive experimentation.



JOURNAL OF COMPUTER AND SYSTEM SCIENCES **31**, 182–209 (1985)

Probabilistic Counting Algorithms for Data Base Applications

PHILIPPE FLAJOLET

INRIA, Rocquencourt, 78153 Le Chesnay, France

AND

G. NIGEL MARTIN

IBM Development Laboratory, Hursley Park, Winchester, Hampshire SO212JN, United Kingdom

Received June 13, 1984; revised April 3, 1985

This paper introduces a class of probabilistic counting algorithms with which one can estimate the number of distinct elements in a large collection of data (typically a large file stored on disk) in a single pass using only a small additional storage (typically less than a hundred binary words) and only a few operations per element scanned. The algorithms are based on statistical observations made on bits of hashed values of records. They are by construction totally insensitive to the replicative structure of elements in the file; they can be used in the context of distributed systems without any degradation of performances and prove especially useful in the context of data bases query optimisation. © 1985 Academic Press, Inc.

1. INTRODUCTION

As data base systems allow the user to specify more and more complex queries, the need arises for efficient processing methods. A complex query can however generally be evaluated in a number of different manners, and the overall performance of a data base system depends rather crucially on the selection of appropriate *decomposition strategies* in each particular case.

Even a problem as trivial as computing the intersection of two collections of data A and B lends itself to a number of different treatments (see, e.g., [7]):

$A \cap B$: 1. Sort A, search each element of B in A and retain it if it appears in A;

2. sort A, sort B, then perform a merge-like operation to determine the intersection;

3. eliminate duplicates in A and/or B using hashing or hash filters, then perform Algorithm 1 or 2.

Each of these evaluation strategy will have a cost essentially determined by the number of records a, b in A and B, and the number of *distinct* elements α, β in A and B, and for typical sorting methods, the costs are:

182

0022-0000/85 $3.00

Open questions

- Better space-accuracy tradeoffs?
- Support other operations?

> *"IT IS QUITE CLEAR that other observable regularities on hashed values of records could have been used…*
>
> *– Flajolet and Martin*

Small sample of work on related problems

1970	Bloom	set membership
1984	Wegman	unbiased sampling estimate
1996–	many authors	refinements (stay tuned)
2000	Indyk	L1 norm
2004	Cormode–Muthukrishnan	frequency estimation deletion and other operations
2005	Giroire	fast stream processing
2012	Lumbroso	full range, asymptotically unbiased
2014	Helmi–Lumbroso–Martinez–Viola	uses neither sampling nor hashing

Cardinality Estimation

- Rules of the game
- Probabilistic counting
- Stochastic averaging
- **Refinements**
- Final frontier

OF

logs and loglogs

To improve space-time tradeoffs, we need to *carefully count bits*.

Relevant quantities
- N is the number of items in the data stream.
- $\lg N$ is the number of bits needed to represent numbers less than N in binary.
- $\lg \lg N$ is the number of bits needed to represent numbers less than $\lg N$ in binary.

For most applications
- N is less than 2^{64}.
- $\lg N$ is less than 64.
- $\lg \lg N$ is less than 7.

Typical PCSA implementations
- Could use $M \lg N$ bits, in theory.
- Use 64-bit words to take advantage of machine-language efficiencies.
- Use (therefore) 64***64** = 4096 bits with $M = 64$ (for 10% accuracy with $N < 2^{64}$).

We can do better (in theory)

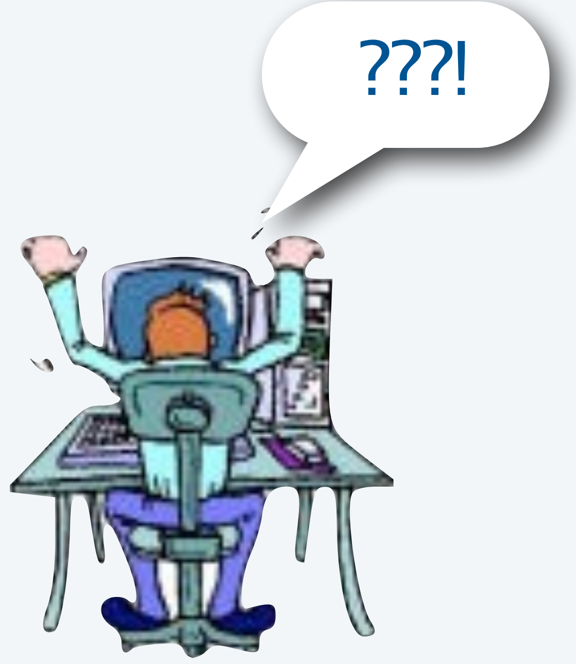
Alon, Matias, and Szegedy
The Space Complexity of Approximating the Frequency Moments
STOC 1996; JCSS 1999.

Contributions
- Studied problem of estimating higher moments
- Formalized idea of *randomized* streaming algorithms
- Won Gödel Prize in 2005 for "foundational contribution"

Theorem (paraphrased to fit context of this talk).
With strongly universal hashing, PC, for any $c > 2$,
- *Uses $O(\log N)$ bits.*
- Is accurate to a factor of c, *with probability at least $2/c$*.

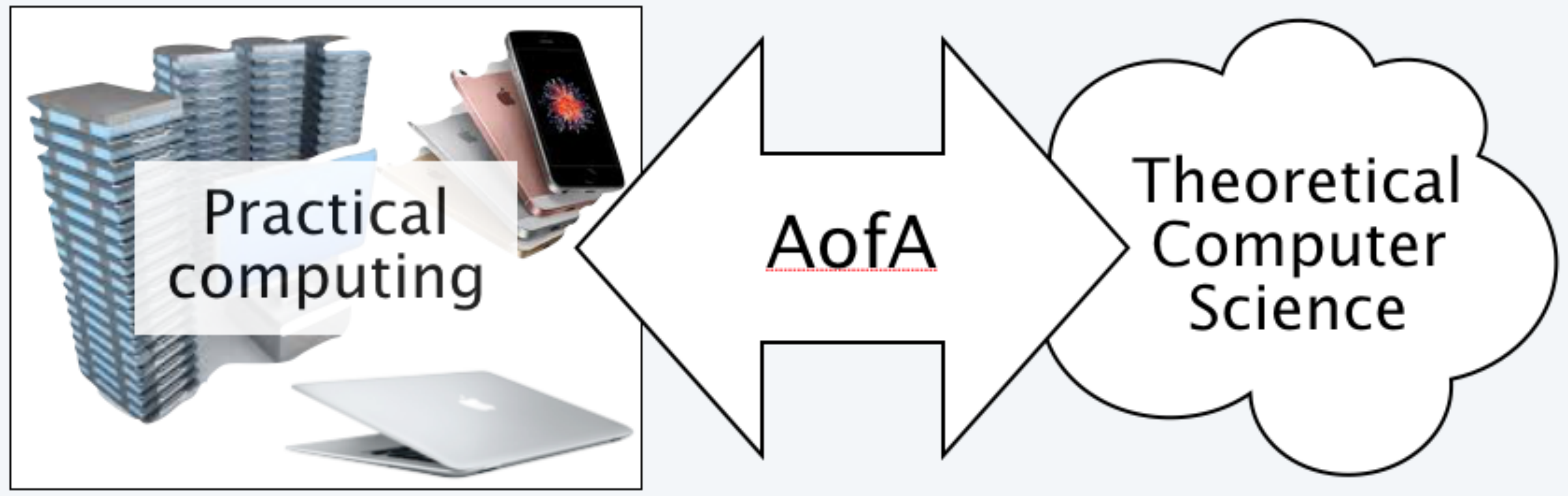
Replaces "uniform hashing" assumption with "random bit existence" assumption

BUT, no impact on cardinality estimation in practice
- "Algorithm" just changes hash function for PC
- Accuracy estimate is too weak to be useful
- No validation



Interesting quote

> *" Flajolet and Martin [assume] that one may use in the algorithm an explicit family of hash functions which exhibits some ideal random properties. Since we are not aware of the existence of such a family of hash functions ..."*
>
> *– Alon, Matias, and Szegedy*



No! That was a thought experiment that they addressed with stochastic averaging.

They also hypothesized that practical hash functions would be as effective as random ones.

They then validated that hypothesis by proving tight bounds that match experimental results.

Points of view re *hashing*

- Theoretical computer science. Uniform hashing assumption is not proved.
- Practical computing. Hashing works for many common data types.
- AofA. *Extensive experiments have validated precise analytic models.*

Points of view re *random bits*

- Theoretical computer science. Random bits exist.
- Practical computing. No, they don't! And randomized algorithms are inconvenient, btw.
- AofA. *More effective path forward is to validate precise analysis even if stronger assumptions are needed.*

We can do better (in theory)

Bar-Yossef, Jayram, Kumar, Sivakumar, and Trevisan

Counting Distinct Elements in a Data Stream

RANDOM 2002.

Contributions

Introduced the idea of using real numbers instead of bit patterns

Improves space-accuracy tradeoff at extra stream-processing expense.

Theorem (paraphrased to fit context of this talk).

With strongly universal hashing, there exists an algorithm that

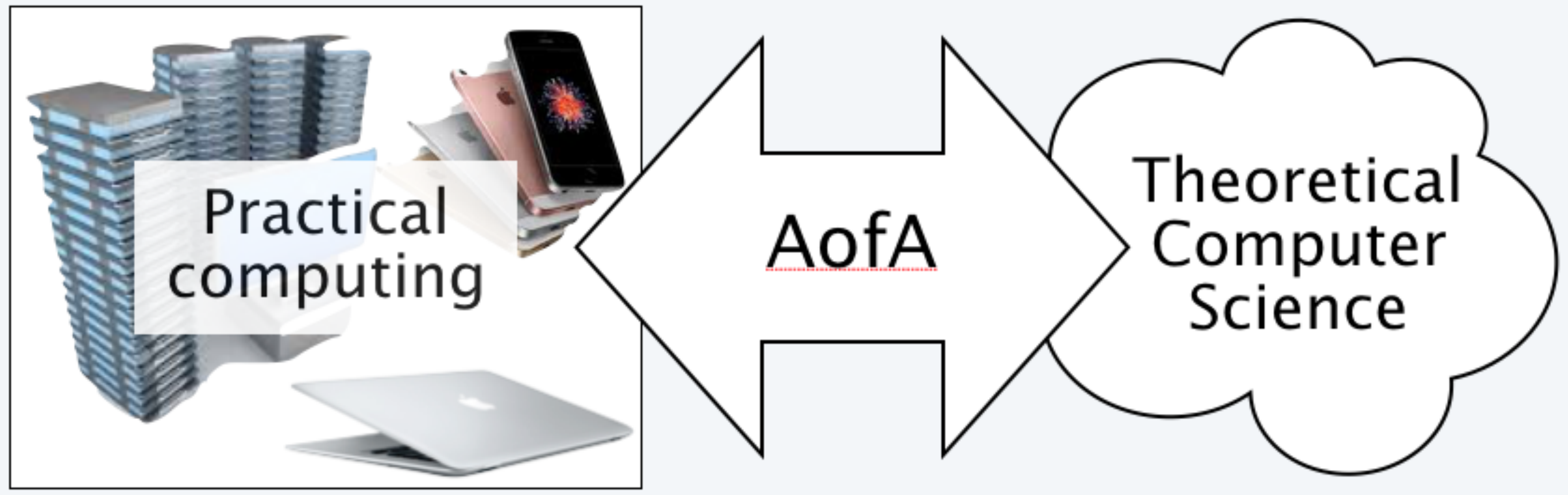
- *Uses O(M log log N) bits.* ← *PCSA uses M lg N bits*
- Achieves relative accuracy $O(1/\sqrt{M})$.

STILL no impact on cardinality estimation in practice

- Infeasible because of high stream-processing expense.
- Big constants hidden in O-notation
- No validation

We can do better (in theory *and* in practice)

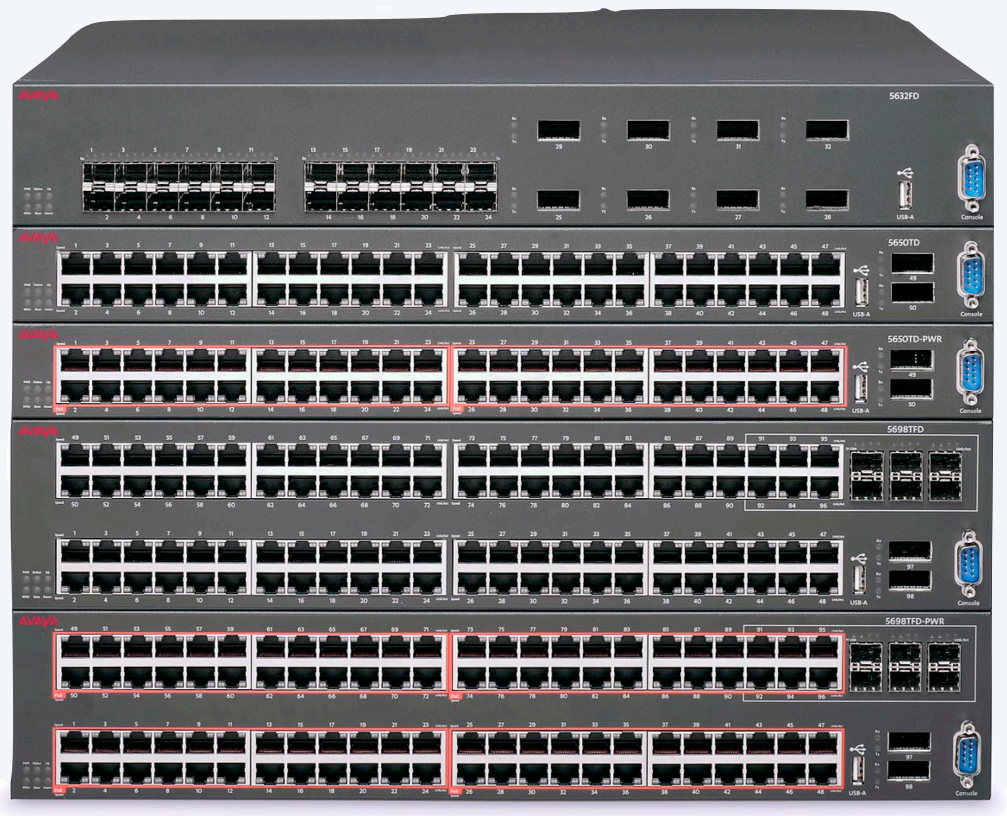
Durand and Flajolet
LogLog Counting of Large Cardinalities
ESA 2003; LNCS volume 2832.



Contributions (independent of BYJKST)
- Presents **LogLog** algorithm, an easy variant of PCSA
- Improves space-accuracy tradeoff *without* extra expense per value
- Full analysis, fully validated with experimentation

Idea. Keep track of min($r(x)$) for each stream.
- lg N bits can save a value (PCSA)
- lg lg N bits can save a bit index in a value

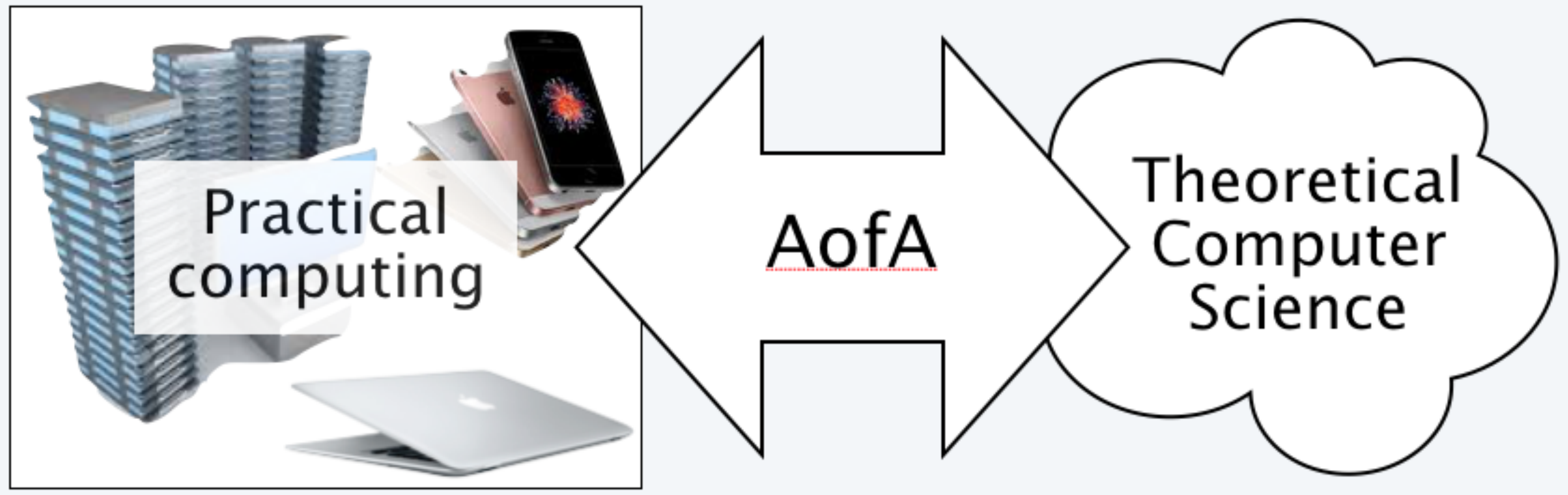
Theorem (paraphrased to fit context of this talk).
*Under the uniform hashing assumption, **LogLog***
- *Uses M lg lg N bits.*
- *Achieves relative accuracy close to* $1.30/\sqrt{M}$.



Practical impact. Deployed for network switches in a telecommunications system.

We can do better (in theory *and* in practice)

Flajolet, Fusy, Gandouet, and Meunier
HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm, *AofA 2007; DMTCS 2007*.



Idea. Use *harmonic* mean to dampen effect of outliers.

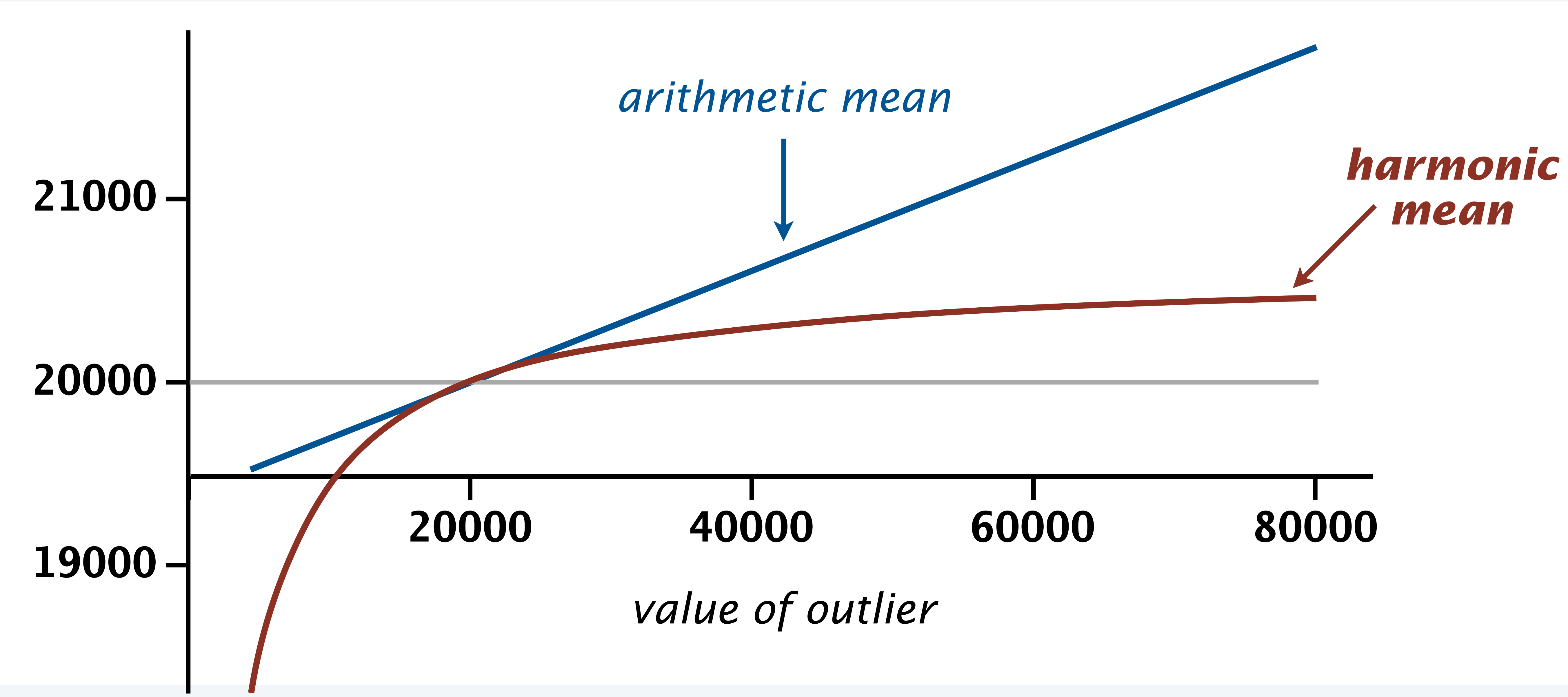
Arithmetic mean

$$\frac{X_1 + X_2 + X_3 + \ldots + X_M}{M}$$

Harmonic mean

$$\frac{M}{\frac{1}{X_1} + \frac{1}{X_2} + \frac{1}{X_3} + \ldots + \frac{1}{X_M}}$$

Illustrative example. 31 values equal to 20000
1 outlier varying between 5000 and 80000



Practical impact. Full analysis and validation allows *immediate* deployment in applications.

We can do better: HyperLogLog algorithm (2007)

```
public static long estimate(Iterable<Long> stream, int M)
{
   int[] bytes = new int[M];
   for (long x : stream)
   {
      int k = hash2(x, M);
      if (bytes[k] < Bits.r(x)) bytes[k] = Bits.r(x);
   }
   double sum = 0.0;
   for (int k = 0; k < M; k++)
      sum += Math.pow(2, -1.0 - bytes[k]);
 return (int) (alpha * M * M / sum);
}
```

Flajolet-Fusy-Gandouet-Meunier 2007

Ideas.

- Use stochastic averaging.
- Keep track of min($r(x)$) for each stream.
- Use harmonic mean.

lg lg N bits

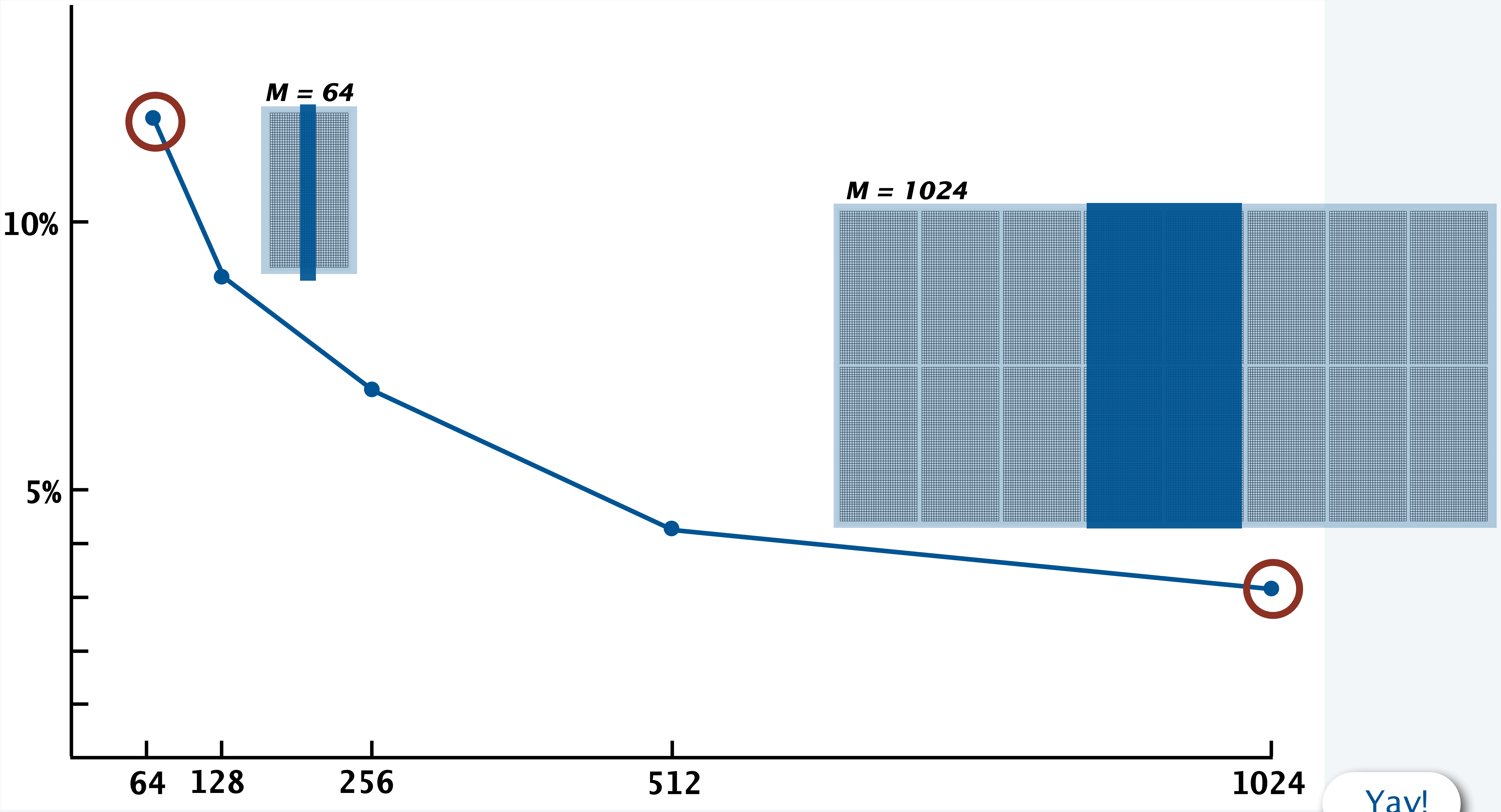
Theorem (paraphrased to fit context of this talk).

Under the uniform hashing assumption, ***HyperLogLog***

- *Uses M log log N bits.*
- *Achieves relative accuracy close to* $1.02/\sqrt{M}$.

Space-accuracy tradeoff for HyperLogLog

Relative accuracy: $\dfrac{1.02}{\sqrt{M}}$



Bottom line (for N < 2^{64}).

- Attain 12.5% relative accuracy with a sketch consisting of 64x6 = $\boxed{\text{396 } \textit{bits}}$.
- Attain 3.1% relative accuracy with a sketch consisting of 1024x6 = 6144 *bits*.

PCSA vs Hyperloglog

Typical PCSA implementations

- Could use $M \lg N$ bits, in theory.
- Use 64-bit words to take advantage of machine-language efficiencies.
- Use (therefore) 64***64** = 4096 bits with $M = 64$ (for 10% accuracy with $N < 2^{64}$).

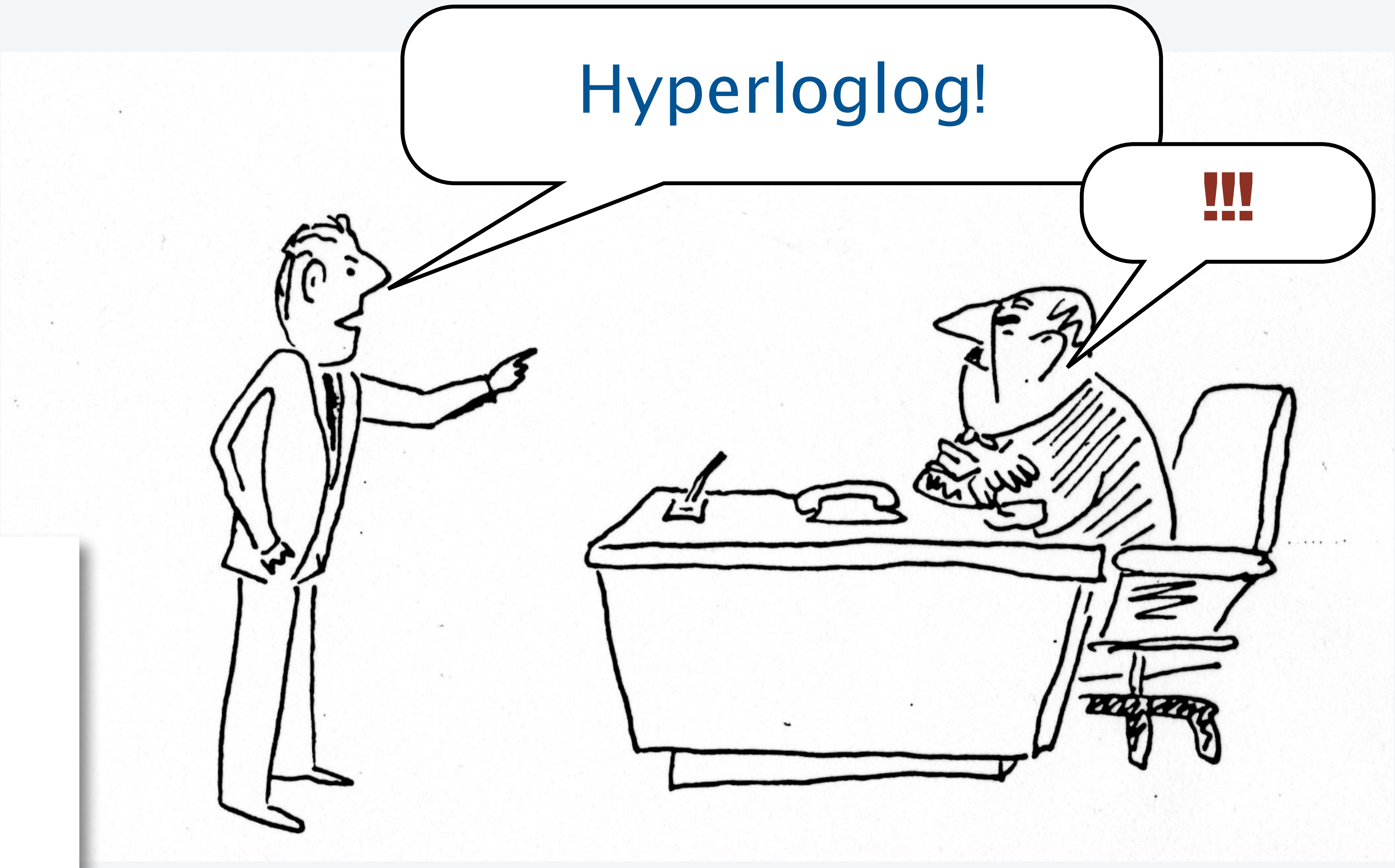
Typical Hyperloglog implementations

- Could use $M \lg \lg N$ bits, in theory.
- Use 8-bit bytes to take advantage of machine-language efficiencies.
- Use (therefore) 64***8** = 512 bits with $M = 64$ (for 10% accuracy with $N < 2^{64}$).

Right answer: Hyperloglog

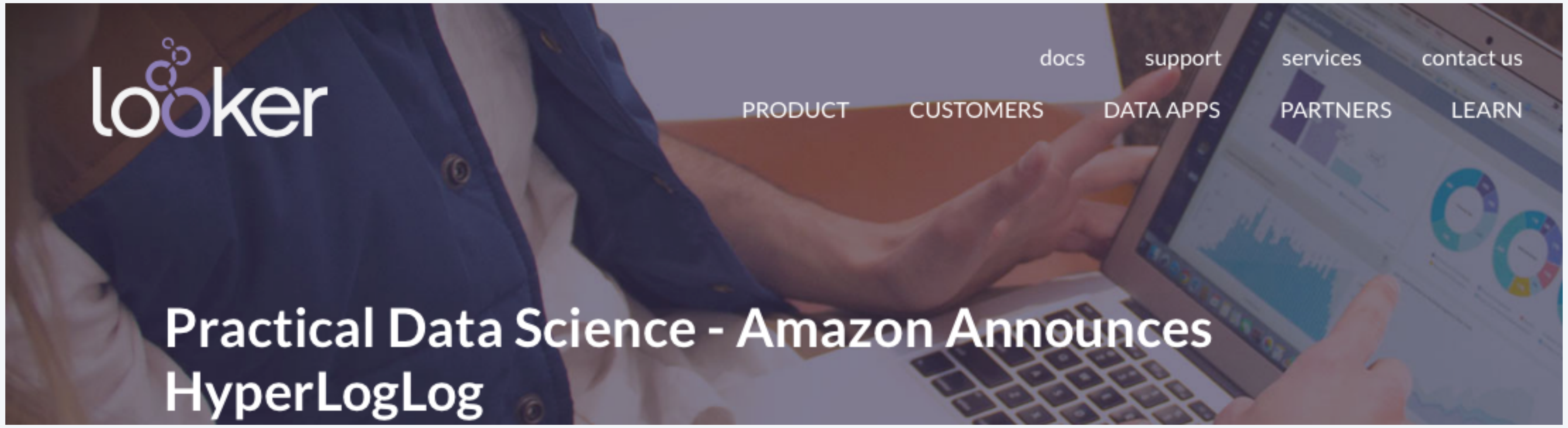
Divide into *M* streams (stochastic averaging)

- Keep track of min(# trailing 1s).
- Use harmonic mean.




```
public static long estimate(Iterable<Long> stream, int M)
{
   byte[] bytes = new byte[M];
   for (long x : stream)
   {
      int k = hash2(x, M);
      if (bytes[k] < Bits.r(x)) bytes[k] = Bits.r(x);
   }
   double sum = 0.0;
   for (int k = 0; k < M; k++)
      sum += Math.pow(2, -1.0 - bytes[k]);
 return (int) (alpha * M * M / sum);
}
```

Validation of Hyperloglog



cradlepoint

neustar

Periscope DATA

AMAZON REDSHIFT

BigQuery

redis

Research at Google

S. Heule, M. Nunkesser and A. Hall
HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm. Extending Database Technology/International Conference on Database Theory 2013.

Philippe Flajolet, mathematician, data scientist, and computer scientist extraordinaire

Philippe Flajolet 1948–2011

For more information about Philippe Flajolet's pioneering contribution to data streaming algorithms, see J. Lumbroso, *How Flajolet Processed Streams with Coin Flips,* https://arxiv.org/abs/1805.00612.

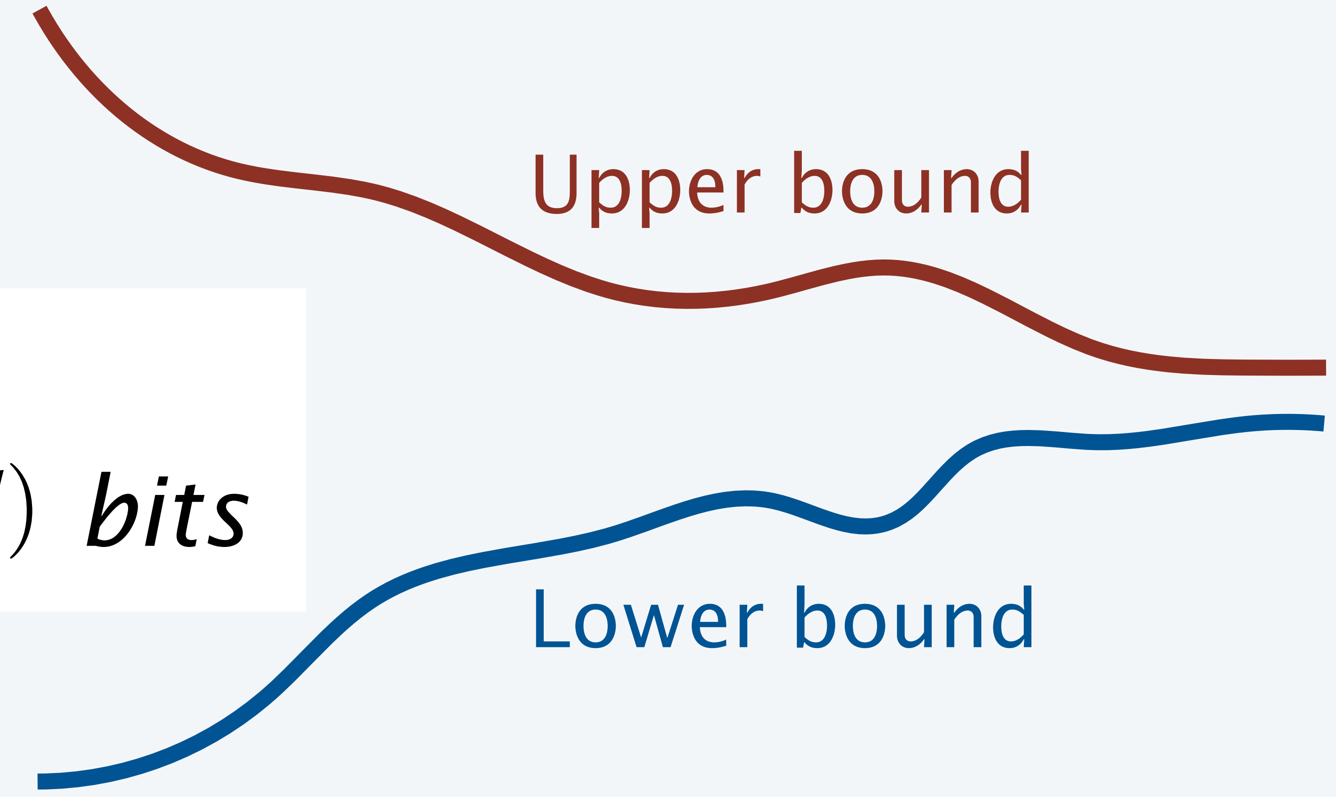
Cardinality Estimation

- Rules of the game
- Probabilistic counting
- Stochastic averaging
- Refinements
- **Final frontier**

We can do a bit better (in theory) *but not much better*

Indyk and Woodruff

Tight Lower Bounds for the Distinct Elements Problem, FOCS 2003.



Theorem (paraphrased to fit context of this talk).

Any algorithm that achieves relative accuracy $O(1/\sqrt{M})$ *must use* $\Omega(M)$ *bits*

loglogN improvement possible

Kane, Nelson, and Woodruff

Optimal Algorithm for the Distinct Elements Problem, PODS 2010.

Theorem (paraphrased to fit context of this talk).

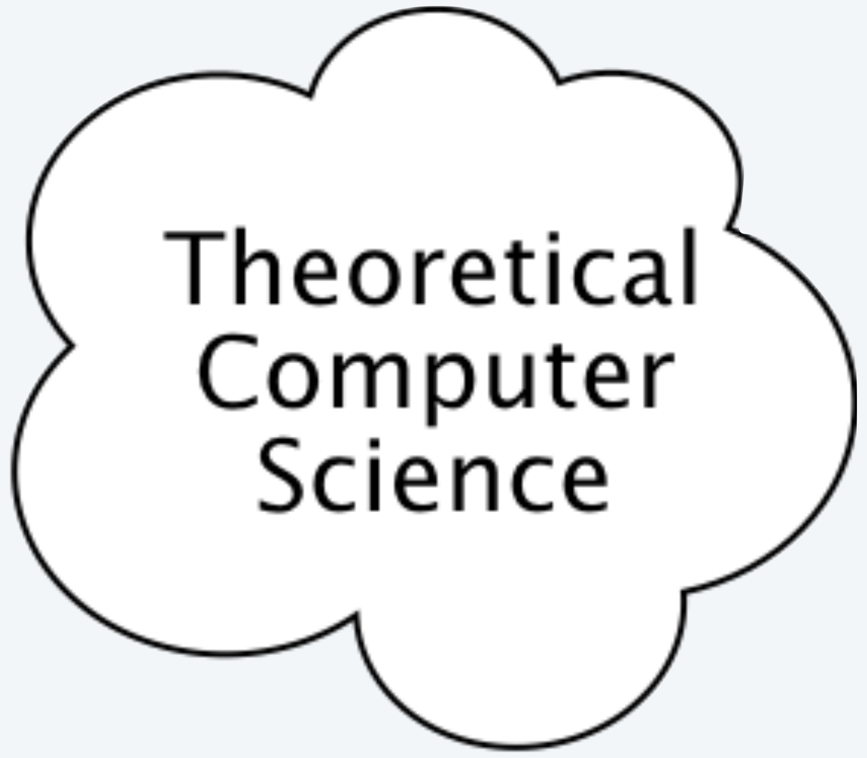
With strongly universal hashing there exists an algorithm that

- *Uses* O(*M*) *bits.*
- *Achieves relative accuracy* $O(1/\sqrt{M})$.

optimal

Unlikely to have impact on cardinality estimation in practice

- Tough to beat HyperLogLog's low stream-processing expense.
- Constants hidden in O-notation not likely to be < 6
- *No validation*



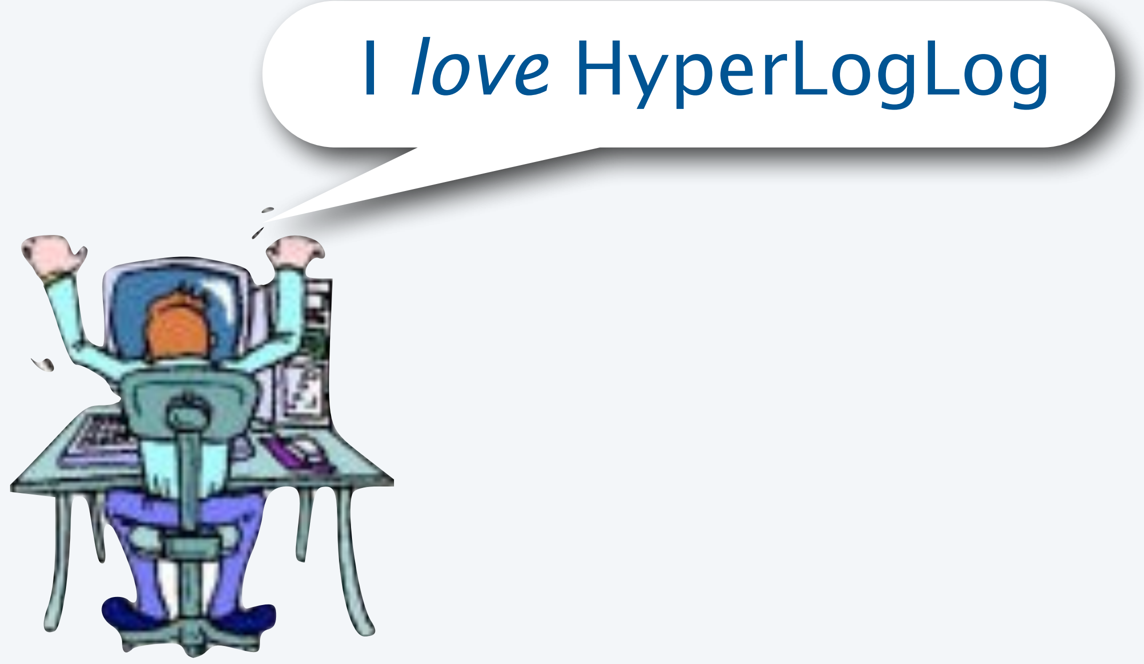
Can we beat HyperLogLog in practice?

Necessary characteristics of a better algorithm

- Makes *one pass* through the stream.
- Uses *a few dozen machine instructions per value*
- Uses *a few hundred bits*
- Achieves 10% relative accuracy or better

> *" I've long thought that there should be a simple algorithm that uses a small constant times M bits..."*
>
> *– Jérémie Lumbroso*

	machine instructions per stream element	*memory bound*	*memory bound when $N < 2^{64}$*	*# bits for 10% accuracy when $N < 2^{64}$*
HyperLogLog	20–30	$M \log\log N$	$6M$	768
BetterAlgorithm	*a few dozen*			*a few hundred*



Also, results need to be validated through extensive experimentation.

A proposal: HyperBitBit (Sedgewick, 2016)

```
public static long estimate(Iterable<String> stream, int M)
{
   int lgN =  5;
   long sketch = 0;
   long sketch2 = 0;
   for (String x : stream)
   {
      long x = hash(s);
      int k = hash2(x, 64);
      if (r(x) > lgN)     sketch  = sketch  | (1L << k);
      if (r(x) > lgN + 1) sketch2 = sketch2 | (1L << k);
      if (p(sketch) > 31)
      { sketch = sketch2; lgN++; sketch2 = 0; }
   }
   return (int) (Math.pow(2, lgN + 5.4 + p(sketch)/32.0));
}
```

bias (determined empirically)

Idea.

- lgN is estimate of $\lg N$
- sketch is 64 indicators whether to increment lgN
- sketch2 is is 64 indicators whether to increment lgN *by 2*
- Update when half the bits in sketch are 1

Initial experiments

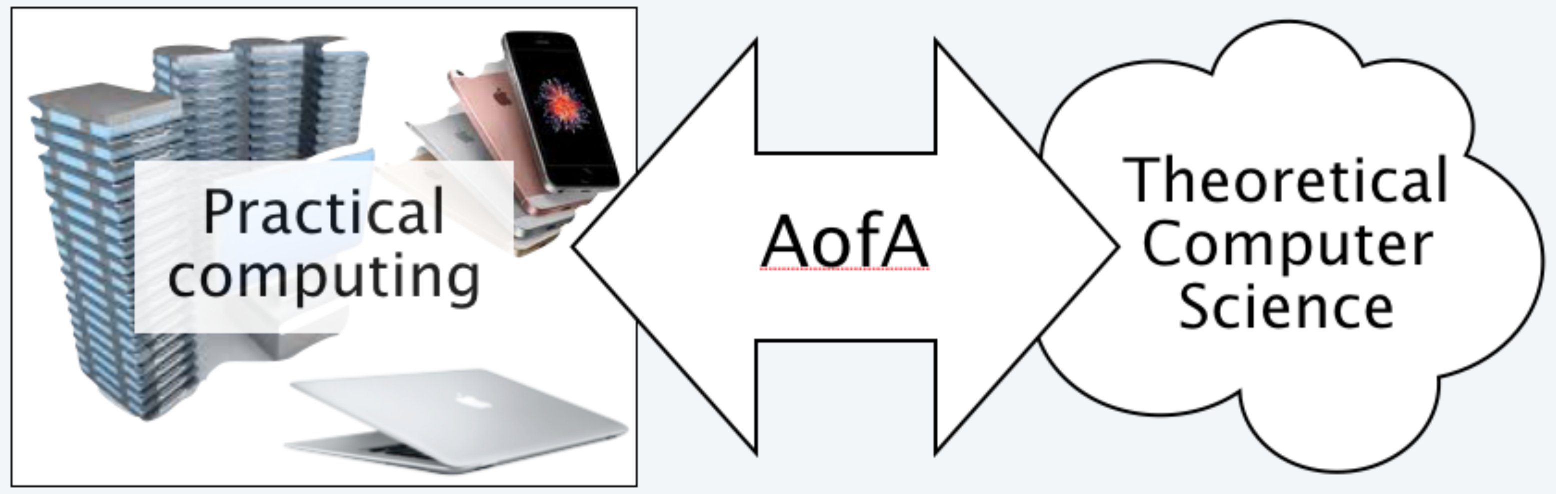
Exact values for web log example

```
% java Hash 1000000 < log.07.f3.txt
242601
% java Hash 2000000 < log.07.f3.txt
483477
% java Hash 4000000 < log.07.f3.txt
883071
% java Hash 6000000 < log.07.f3.txt
1097944
```

HyperBitBit estimates

```
% java HyperBitBit 1000000 < log.07.f3.txt
234219
% java HyperBitBit 2000000 < log.07.f3.txt
499889
% java HyperBitBit 4000000 < log.07.f3.txt
916801
% java HyperBitBit 6000000 < log.07.f3.txt
1044043
```

	1,000,000	2,000,000	4,000,000	6,000,000
Exact	242,601	483,477	883,071	1,097,944
HyperBitBit	234,219	499,889	916,801	1,044,043
ratio	1.05	1.03	0.96	1.03



Conjecture. *On practical data,* ***HyperBitBit****, for* $N < 2^{64}$,

- *Uses* 128 + 6 *bits.*
- *Estimates cardinality within* 10% *of the actual.*

Next steps.

- *Analyze.*
- *Experiment.*
- *Iterate.*

Summary for cardinality estimation algorithms

	time bound	*memory bound (bits)*	*# bits for 10% accuracy for 1 billion inputs*
Wrong answer	N^2	$N \lg N$	64 billion
Sort and count	$N \log N$	$N \lg N$	64 billion
Existence table	N	U	1 billion
Hash table	N^*	$N \lg N$	64 billion
PCSA	N^*	$M \lg N$	4096
HyperLogLog	N^*	$M \lg\lg N$	512
HyperBitBit ?	N^*	$2M + \lg\lg N$	134

A Case Study: Cardinality Estimation

- Exact cardinality count
- Probabilistic counting
- Stochastic averaging
- Refinements

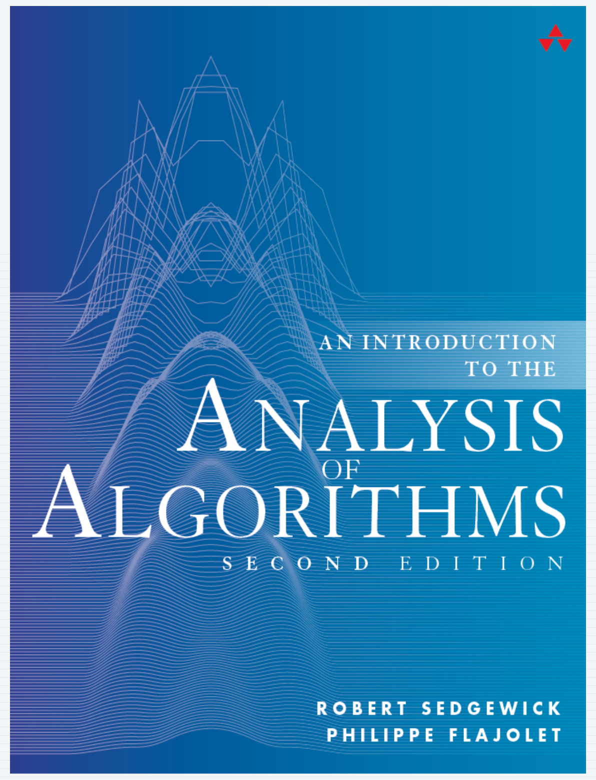
OF

Why study analysis of algorithms and analytic combinatorics?

HyperLogLog is a case in point.

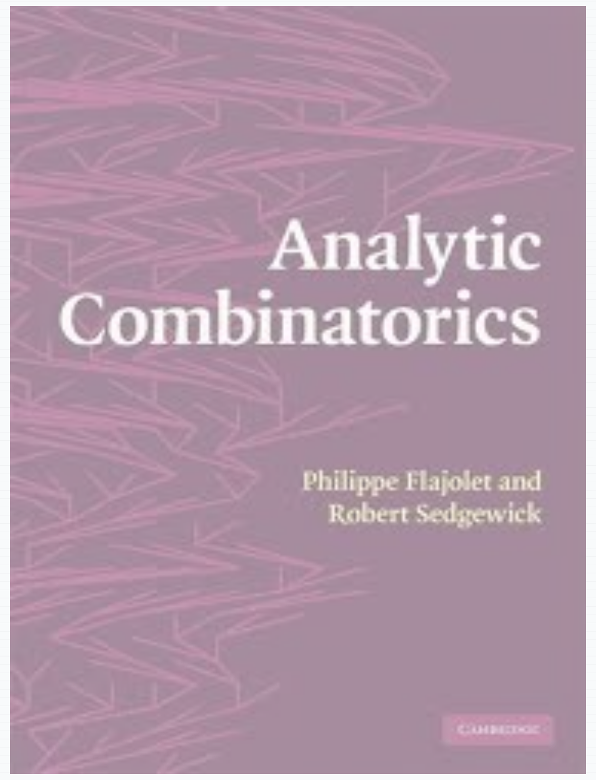
- Impact is broad and far-reaching. ✓
- Old roots, new opportunities. ✓
- Allows solution of otherwise unsolvable problems. ✓
- Intellectually stimulating. ✓
- Implementations teach programming proficiency. ✓
- May unlock the secrets of life and of the universe. *who knows?*
- Useful for fun and profit. ✓

HyperLogLog



Analysis of Algorithms

Original MOOC title: ANALYTIC COMBINATORICS, PART ONE



Analytic Combinatorics

Original MOOC title: ANALYTIC COMBINATORICS, PART TWO

http://aofa.cs.princeton.edu

http://ac.cs.princeton.edu