FOUNTAINHEAD

"Big Data Algorithms"

~ Adventures in Time & Space ~

Baruch College

Tuesday 24th January 2017

FOUNTAINHEAD

Feynman Algorithm

1. Write down the problem.
2. Think really hard.
3. Write down the solution.

FOUNTAINHEAD

Right versus “Good Enough” Answers

FOUNTAINHEAD

Speed versus Accuracy

Generally speaking, there is always a tradeoff between speed and accuracy.

FOUNTAINHEAD

51% Right = Genius?

Chief Technology Officer:

Anything less than 100% correct is worthless.

Chief Quant:

If I am systematically correct 51% of the time, I'm a genius!

FOUNTAINHEAD

Intelligent Bets

It's not how often you win or lose, but rather how much you win when you win and how much you lose when you lose.

True whether you are talking about trading, investing *or* technology!

FOUNTAINHEAD

Moore’s Law and Algorithms

FOUNTAINHEAD

Moore's Law

Moore's law states (Gordon Moore, Intel co-founder, 1961):

"The density of transistors in integrated circuits will double every 1 to 2 years."

The consequence has been that the computational power of CPUs has doubled every 18 months for decades. But the "free lunch" is over.

FOUNTAINHEAD

Algorithms

But what is not widely recognized is that:

"In the same period, performance gains from improvements in algorithms have vastly exceeded Moore's Law."

FOUNTAINHEAD

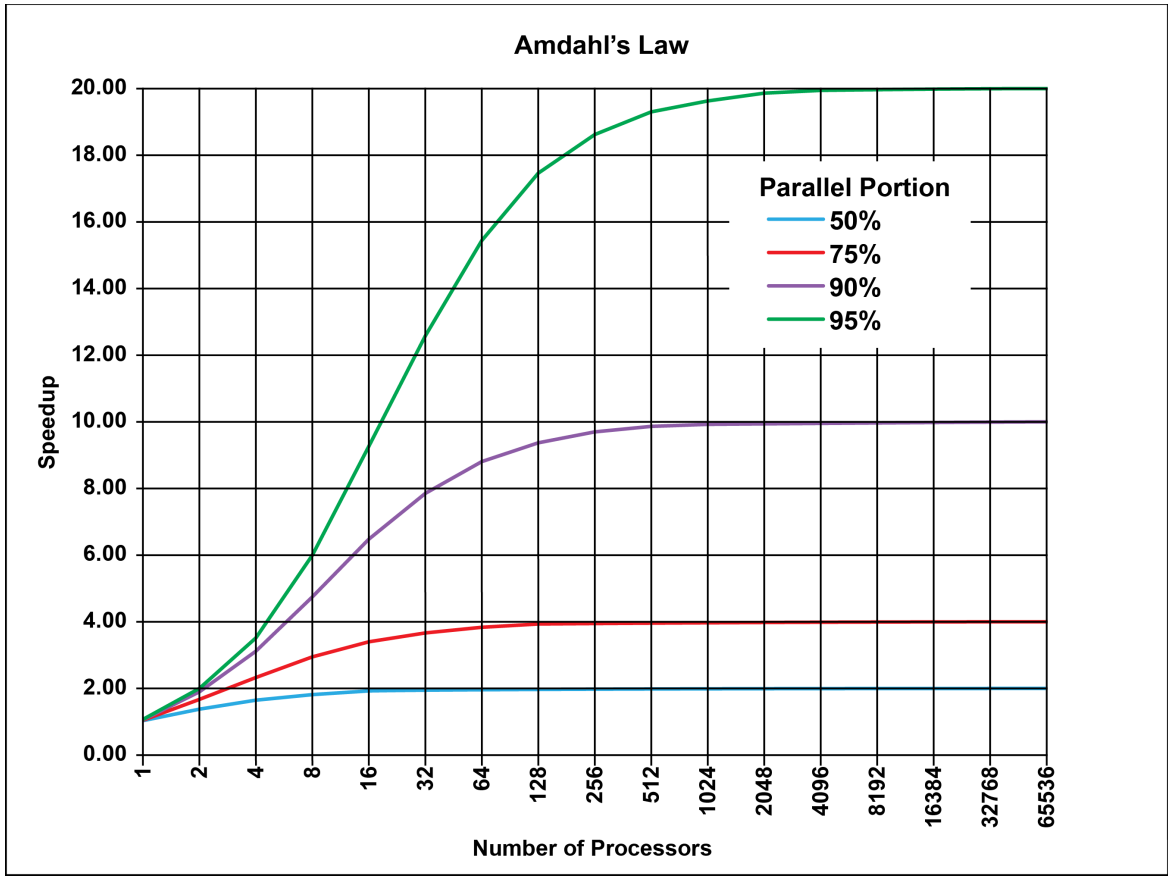
Amdahl's Law

$$S(n) = \frac{1}{a + (1-a)/n}$$

NB: Assumes fixed problem size.

FOUNTAINHEAD

Amdahl's Law



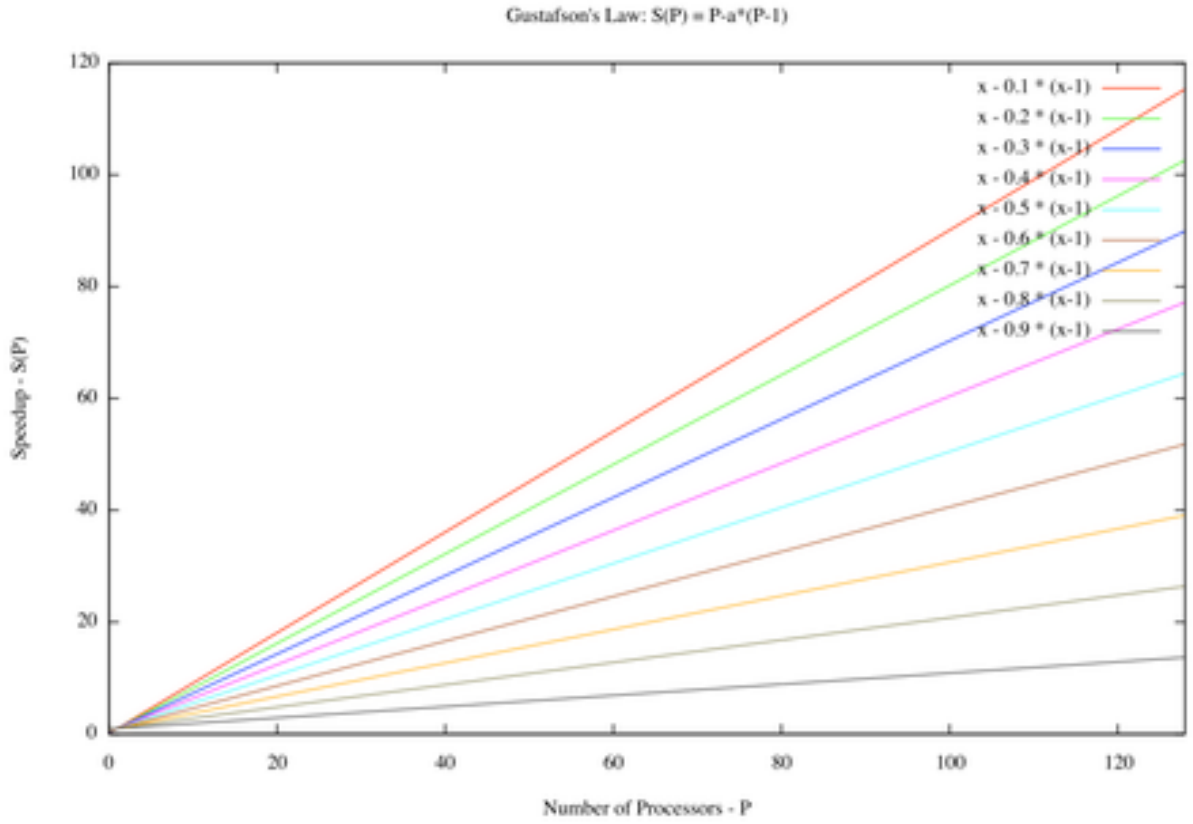
FOUNTAINHEAD

Gustafson–Barsis' Law

$$S(n) = n - a(n-1)$$

FOUNTAINHEAD

Gustafson–Barsis' law



FOUNTAINHEAD

Parallel Algorithms

"The vast majority of algorithms we have are serial in nature. Lot's of gains still to be made from parallel algorithms."

FOUNTAINHEAD

Approximations & Heuristics

There are lots of problems for which we don’t have good exact algorithms, but good success with approximate algorithms and heuristics.

FOUNTAINHEAD

Big-O Notation

FOUNTAINHEAD

First, a Definition of Big-O

Definition:

Big-O notation is used to classify the behavior of algorithms by how they respond in terms of processing time or working space requirements to changes in the size of data input.

FOUNTAINHEAD

Big-O Examples

Some Big-O examples for running times (memory/disk/network):

O(1)	constant time	[SHORTER]
O(log(n))	logarithmic time	
O((log(n))^c)	polylogarithmic time	
O(n)	linear time	
O(n^2)	quadratic time	
O(n^c)	polynomial time	
O(c^n)	exponential time	[LONGER]

FOUNTAINHEAD

The Need for Big-O Notation

Reason to use Big-O notation:

- Time needed to carry out a task is proportional to the number of basic operations; for compute they are arithmetic operations, tests, data read/write and so on.
- When these operations change in response to a program's input, we need a way quantify that change, particularly in the limit of large input.
- We are usually interested in the *worst case* in the limit. That's Big-O.

FOUNTAINHEAD

Compendium of Big-O for Common Algorithms

See cheatsheet on MFE forum.

FOUNTAINHEAD

Sublinearity

Characteristics of sublinearity from an algorithmic perspective:

- Data too big or too little time available? [Sublinear time]
  - Read only part of the data. Sublinear in time. E.g. Sampling.
- Data too big to fit into main memory? [Sublinear space]
  - Store on disk with block access. Sublinear in I/O.
  - Throw some of the data away. Sublinear in space. E.g. Stream.
- Data too big to fit on a single machine. [Sublinear communication]
  - Distribute data and collaborate via communication. Sublinear in communication. E.g. MapReduce.

FOUNTAINHEAD

Sublinearity

So what is the key concept behind sublinearity?

Time / space / communication spent is of O(input size).

FOUNTAINHEAD

P versus NP-Problems

The P versus NP problem is a major unsolved problem in computer science. Informally, it asks whether every problem whose solution can be quickly verified by a computer can also be quickly solved by a computer.

FOUNTAINHEAD

Approximations and Heuristics

Approximations and heuristics can be always be used to find an approximate solution in less time that an accurate algorithm. However, for NP type problems approximations and heuristics are the *only* available tools to find an answer.

FOUNTAINHEAD

CUNY Supercomputer

FOUNTAINHEAD

CUNY Supercomputer

Walkthrough requesting an account.

FOUNTAINHEAD

MapReduce and Hadoop

Fountainhead

MapReduce

What is “MapReduce”?

MapReduce was introduced by Google in 2004. It is a Framework for distributed computing of large data sets on clusters of commodity computers.

In essence it follows the principles of “divide and conquer” by dividing data into many parts that can then be worked on independently by many processes in parallel.

FOUNTAINHEAD

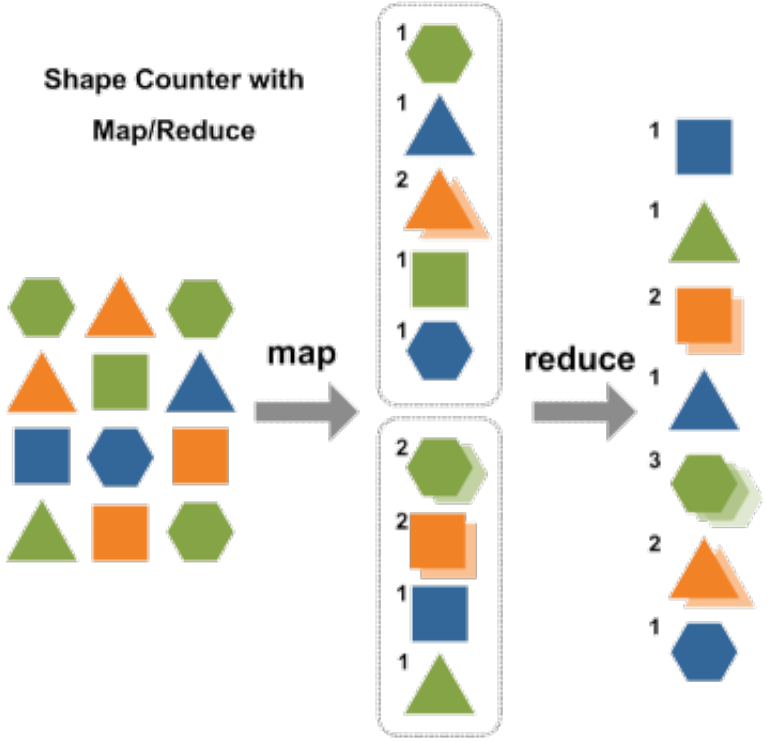
MapReduce

A brief history:

- Created by Doug Cutting (with help from others).
- Precursor was a project called Nutch. Web search.
- 2004 GFS and MapReduce papers published.
- 2004-2006 DFS and MapReduce added to Nutch.
- 2006-2008 Doug Cutting hired Yahoo! Hadoop spun out of Nutch and named after Cutting's son's toy elephant.

Fountainhead

MapReduce



Fountainhead

Hadoop

From a physical point of view Hadoop is a cluster:

- Lots of disks, spinning all the time.
- Redundancy, since disks die.
- Lots of CPU cores, working all the time.
- Retry, since network errors happen.
- Scalable in the sense that nodes can join and leave the cluster at any time.

Fountainhead

Hadoop

From a systems point of view Hadoop is:

- Scalable - many servers with lots of cores and spindles.
- Reliable - detect failures, redundant storage.
- Fault-tolerant - auto-retry, self-healing.
- Simple - use many servers as one really big computer.

Fountainhead

Hadoop

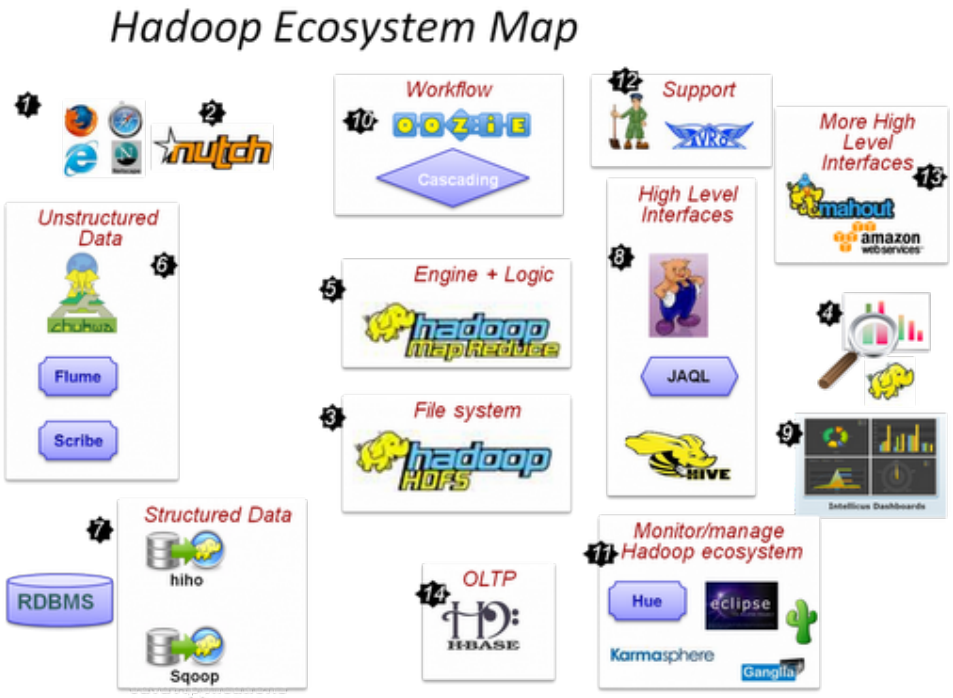
From a software point of view Hadoop is:

- Implements the MapReduce framework.
- Responsible for dividing the data a running jobs in parallel on many servers.
- Handles re-trying a task that fails and validating complete results.
- Aims to make “map” and “reduce” operations easy.

Fountainhead

Hadoop

From an ecosystem point of view Hadoop is:



Fountainhead

Hadoop Tools

I like to think of Hadoop as the hub of a wheel, and the ecosystem of tools that surround it as the spokes of that wheel. Typically the spokes keep the rim of the wheel in contact with the road (data) which in turns the hub (Hadoop). Hadoop now has quite a large number of tools that aim to extend and simplify Hadoop for the user. Let's look at some of the most important.

Fountainhead

Pig

Pig is a tool that sits on top of Hadoop:

- Consists of two parts:
  - A data flow language, called appropriately “pig latin”.
  - An execution environment for pig latin.
- It is a way to process data at a higher level than raw MapReduce.

Fountainhead

Hadoop – Hive

Hive is a tool that sits on top of Hadoop:

- It is a framework for data warehousing.
- Integrates SQL into the mix. SQL-like language HiveQL. Providing …
- Data summarization.
- Data query.
- Data analysis.

Fountainhead

Hadoop – Hbase

HBase is a tool that sits on top of Hadoop:

- It is a distributed column-oriented database.
- Often used when near real-time read/write random-access to data is needed.
- Modeled after Google's "Big Table", which can be thought of as a sparse, distributed, multi-dimensional, sorted map. (Pause to explain what that means!)

Fountainhead

Hadoop – Zookeeper

Zookeeper provides a range of services:

- Distributed configuration service.
- Synchronization service.
- Naming registry.
- … for large distributed systems.
- (It’s name is well chosen because it helps herd an otherwise unruly group of activities.)

Fountainhead

Hadoop – Sqoop

Sqoop is a tool to:

- Import data from relational databases into Hadoop.
- Sqoop uses JDBC to connect to a database.
- Examines the database schema and generates the data structures to store data in Hadoop.
- Uses MapReduce to transfer data in parallel.
- (Think “scoop up the data”.)

Fountainhead

Hadoop - Other Tools

And there are many others:

- Howl - a table management service.
- Oozie - a workflow engine.
- Mahout - machine learning.
- Chukwa - large-scale log collection and analysis.
- Avro - format for data interchange.
- … and a host of other projects.

FOUNTAINHEAD

Monte-Carlo on GPUs

FOUNTAINHEAD

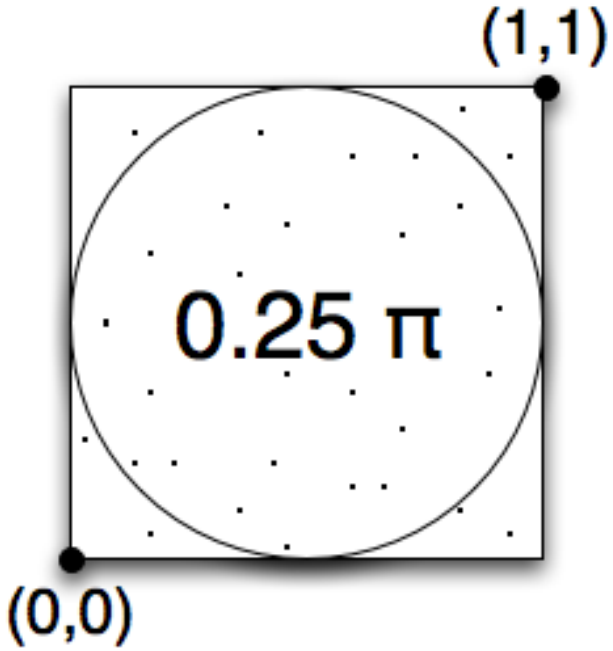
Example: Monte-Carlo using CUDA Thrust

Let's consider a simple example of how Monte-Carlo can Be mapped onto GPUs using CUDA Thrust.

CUDA Thrust is a C++ template library that is part of the CUDA toolkit and has containers, iterators and algorithms; and is particularly handy for doing Monte-Carlo on GPUs.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)



This is a very simple example that estimates the value of the constant **`PI`** while illustrating the key points when doing Monte-Carlo on GPUs.

(As an aside, it also demonstrates the power of CUDA Thrust.)

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
int main() {
    size_t N = 50000000; // Number of Monte-Carlo simulations.
    // DEVICE: Generate random points within a unit square.
    thrust::device_vector<float2> d_random(N);
    thrust::generate(d_random.begin(), d_random.end(), random_point());
    // DEVICE: Flags to mark points as lying inside or outside the circle.
    thrust::device_vector<unsigned int> d_inside(N);
    // DEVICE: Function evaluation. Mark points as inside or outside.
    thrust::transform(d_random.begin(), d_random.end(),
                      d_inside.begin(), inside_circle());
    // DEVICE: Aggregation.
    size_t total = thrust::count(d_inside.begin(), d_inside.end(), 1);
    // HOST: Print estimate of PI.
    std::cout << "PI: " << 4.0*(float)total/(float)N << std::endl;
    return 0;
}
```

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
struct random_point {
    __device__
    float2 operator()() {
        thrust::default_random_engine rng;
        return make_float2(
                (float)rng() / thrust::default_random_engine::max,
                (float)rng() / thrust::default_random_engine::max);
    }
};
```

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
struct inside_circle {
    __device__
    unsigned int operator()(float2 p) const {
    return (((p.x-0.5)*(p.x-0.5)+(p.y-0.5)*(p.y-0.5))<0.25) ? 1 : 0;
    }
};
```

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

Let's look at the code and how it relates to the steps (elements) of Monte-Carlo.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
// DEVICE: Generate random points within a unit square.
thrust::device_vector<float2> d_random(N);
thrust::generate(d_random.begin(), d_random.end(), random_point());
```

STEP 1: Random number generation. Key points:

- Random numbers are generated in parallel on the GPU.
- Data is stored on the GPU directly, so co-locating the data with the processing power in later steps.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

STEP 2: Generate simulation data. Key points:

- In this example, the random numbers are used directly and do not need to be transformed into something else.
- If higher level simulation data is needed, then the same principles apply: ideally, generate it on the GPU, store the data on the device, and operate on it in-situ.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
// DEVICE: Flags to mark points as lying inside or outside the circle.
thrust::device_vector<unsigned int> d_inside(N);
// DEVICE: Function evaluation. Mark points as inside or outside.
thrust::transform(d_random.begin(), d_random.end(),
                  d_inside.begin(), inside_circle());
```

STEP 3: Function evaluation. Key points:

- Function evaluation is done on the GPU in parallel.
- Work can be done on the simulation data in-situ because it was generated & stored on the GPU directly.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
// DEVICE: Aggregation.
size_t total = thrust::count(d_inside.begin(), d_inside.end(), 1);
// HOST: Print estimate of PI.
std::cout << "PI: " << 4.0*(float)total/(float)N << std::endl;
```

STEP 4: Aggregation. Key points:

- Aggregation is done on the GPU using parallel constructs and highly GPU-optimized algorithms (courtesy of Thrust).
- Data has been kept on the device throughout and only the final result is transferred back to the host.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

Key takeaways from this example:

- Use the tools! CUDA Thrust is a very powerful abstraction tool for doing Monte-Carlo on GPUs.
- It's efficient too, as it generates GPU optimized code.
- Do as much work on the data as possible in-situ, and in parallel. Only bring back to the host the minimum you need to get an answer.

FOUNTAINHEAD

Random Thoughts and Techniques

FOUNTAINHEAD

Sketching and Streaming

Basics of sketching data structures:

- Sketching data structures store a summary of a data set when storing the whole would be prohibitive (c.f. sublinear in space).
- Answer questions about data extremely efficiently at the price of an occasional error. But error *quantified*.
- No errors acceptable → sketching data structure collapses to regular data structure.

FOUNTAINHEAD

Counting Distinct Elements

Input: Stream of numbers (say in range [1,n])

Example: 3 4 3 2 4 17 4 11 3 3 …

Goal: Compute number of distinct elements.

Here 5 because we saw {3,4,2,17,11}

Simple Solution: Just maintain a list of items seen thus far

Stream: 3 4 3 2 4 17 4 11 3 3 List: { }

FOUNTAINHEAD

Counting Distinct Elements

Input: Stream of numbers (say in range [1,n])

Example: 3 4 3 2 4 17 4 11 3 3 …

Goal: Compute number of distinct elements.

Here 5 because we saw {3,4,2,17,11}

Simple Solution: Just maintain a list of items seen thus far

Stream: 3 4 3 2 4 17 4 11 3 3 List: { 3 }

FOUNTAINHEAD

Counting Distinct Elements

Input: Stream of numbers (say in range [1,n])

Example: 3 4 3 2 4 17 4 11 3 3 …

Goal: Compute number of distinct elements.

Here 5 because we saw {3,4,2,17,11}

Simple Solution: Just maintain a list of items seen thus far

Stream: 3 4 3 2 4 17 4 11 3 3 List: { 3, 4}

FOUNTAINHEAD

Counting Distinct Elements

Input: Stream of numbers (say in range [1,n])

Example: 3 4 3 2 4 17 4 11 3 3 …

Goal: Compute number of distinct elements.

Here 5 because we saw {3,4,2,17,11}

Simple Solution: Just maintain a list of items seen thus far

Stream: 3 4 3 2 4 17 4 11 3 3 List: { 3, 4}

FOUNTAINHEAD

Counting Distinct Elements

Input: Stream of numbers (say in range [1,n])

Example: 3 4 3 2 4 17 4 11 3 3 …

Goal: Compute number of distinct elements.

Here 5 because we saw {3,4,2,17,11}

Simple Solution: Just maintain a list of items seen thus far

Stream: 3 4 3 2 4 17 4 11 3 3 List: { 3, 4, 2}

FOUNTAINHEAD

Counting Distinct Elements

Note: The algorithm tracks the numbers seen thus far.

Question: What if it can remember only 1 number?
(very limited memory)

Trouble: Can barely remember anything about past.

Stream: 3 4 3 2 4 17 4 11 3 3

When you scan next element, no clue if already seen?

FOUNTAINHEAD

Counting Distinct Elements

Seems completely hopeless ?

Intuition only partly right

Cannot hope to count exactly.

But who cares if answer is 3,425,269 or 3,425,587 ?

Approximate counting possible!!

Technique: Min-hashing

(beautiful use of approximation and randomization)

FOUNTAINHEAD

Counting Distinct Elements

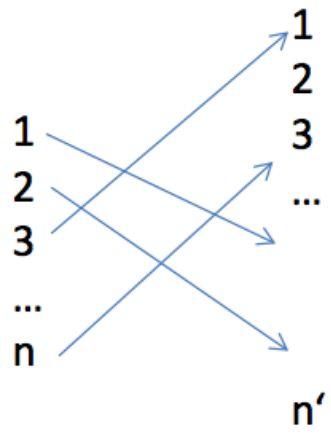
Basic Idea [Flajolet-Martin 82]: Use a random hash function (map).
(e.g. encryption function)

h:[1,n] -> [1,n'] say n' >> n

Algorithm: Keep track of min h(i)

Stream = 2 8 2 7 ...

h(2)



FOUNTAINHEAD

Counting Distinct Elements

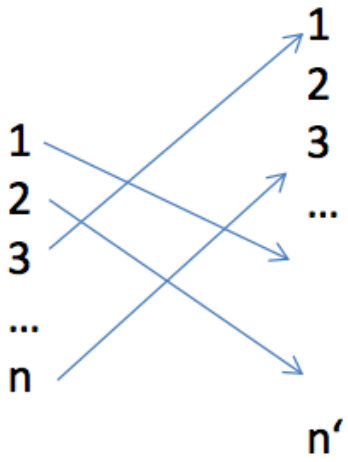
Basic Idea [Flajolet-Martin 82]: Use a random hash function (map).
(e.g. encryption function)

h:[1,n] -> [1,n'] say n' >> n

Algorithm: Keep track of min h(i)

Stream = 2 8 2 7 …

h(8)

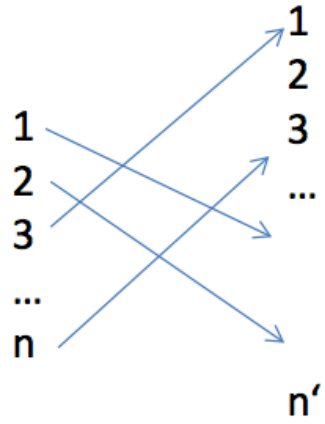


FOUNTAINHEAD

Counting Distinct Elements

Basic Idea [Flajolet-Martin 82]: Use a random hash function (map).
(e.g. encryption function)

h:[1,n] -> [1,n'] say n' >> n



Algorithm: Keep track of min h(i)

Stream = 2 8 2 7 ...

h(8)

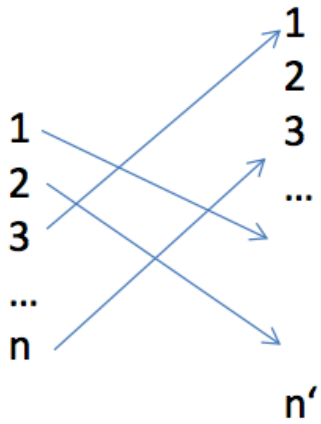
Note: h(i) is same every time i is encountered.

FOUNTAINHEAD

Counting Distinct Elements

Basic Idea [Flajolet-Martin 82]: Use a random hash function (map).
(e.g. encryption function)

h:[1,n] -> [1,n'] say n' >> n



Algorithm: Keep track of min h(i)

Stream = 2 8 2 7 ...

h(8)

Note: h(i) is same every time i is encountered.

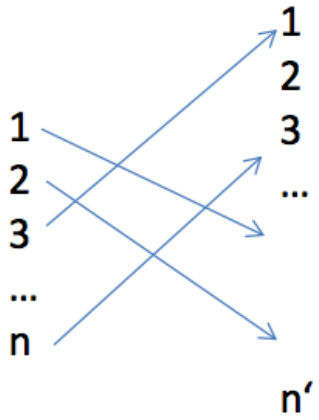
FOUNTAINHEAD

Counting Distinct Elements

Keep track of min h(i)

Suppose 1 distinct element (stream = 1 1 1 ...)

Min h(i) ≈ n'/2



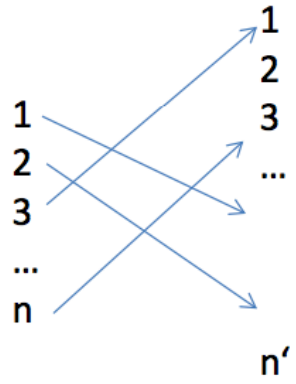
FOUNTAINHEAD

Counting Distinct Elements

Keep track of min h(i)

Suppose 2 distinct elements

Min h(i) ≈ n'/3



If k items seen, expect min-value to be around n'/(k+1)

Solution: Estimate of # elements = n'/ min h(i) – 1

FOUNTAINHEAD

Counting Distinct Elements

Randomness could mess things up.

E.g. May just 1 element, expect min(h(i)) = n'/2
But min(h(i)) could be far.

Standard trick: O(1) such hash functions, take median entry.

Theorem: For any $\epsilon > 0$, can estimate distinct elements to within $1 \pm \epsilon$ factor accuracy with high probability. Space = $O\left(\frac{1}{\epsilon^2}\right)$

FOUNTAINHEAD

Counting Distinct Elements

Random hash function h.
We need that h(i) value be same every time we see i.

One has to store each h(i) some where.
h(1), h(2), h(3),…, h(n) need n log n space??

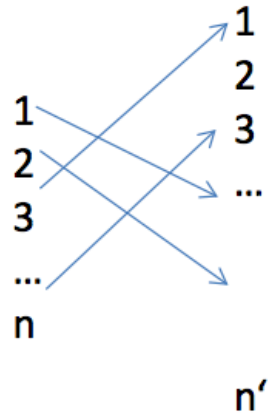
Did we just disguise our inherent problem?

There is a fix!
Key idea: Do not need full power of randomness

FOUNTAINHEAD

Counting Distinct Elements

Do not need full randomness



Pairwise independence: For any a_1, a_2, x_1, x_2

Pr [$h(x_1) = a_1$ and $h(x_2) = a_2$] = $1/n'^2$

Such an h is very simple to store

$h(x) = ax + b \mod (p)$ [just need to store a and b]

FOUNTAINHEAD

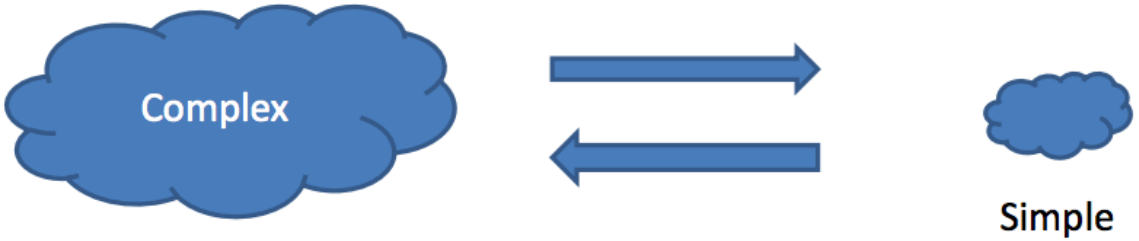
Counting Distinct Elements

Similarity of Web pages (if mostly similar words)

Google: Page -> Few min-hash values (few bits)

Similar page detection: quadratic -> Linear time

Sketching



Tons of amazing applications (several researchers ...)

FOUNTAINHEAD

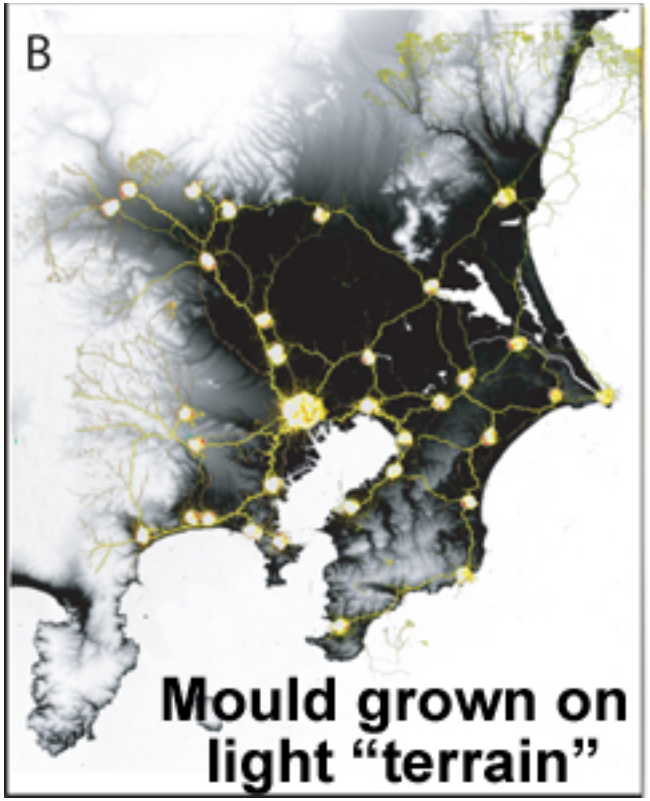
Slime Mold Algorithm

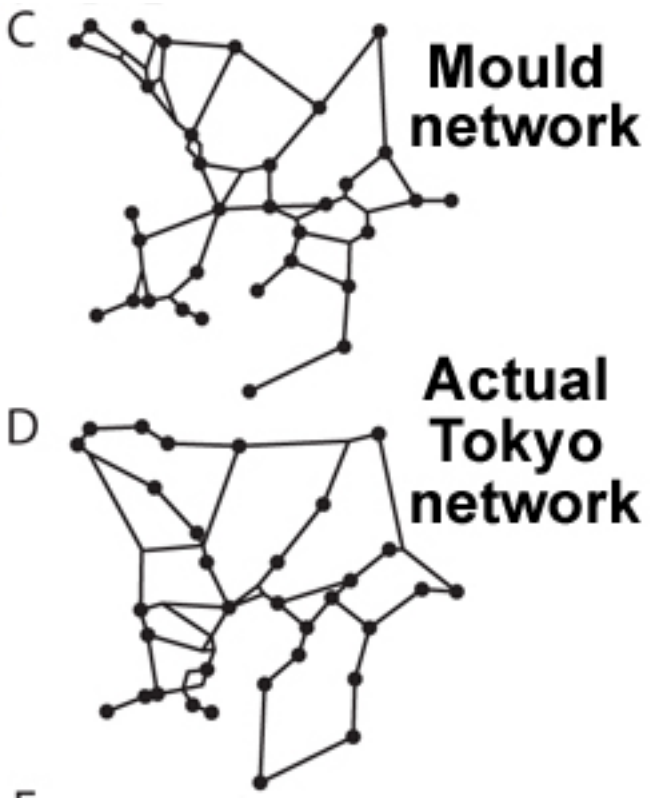
The slime mold algorithm:

- Belongs to a class of algorithms that take their inspiration from nature.
- Through the mold growth process it has been found that slime mold can solve a range of minimization problems.
- Non-random mutation. (*Not* like throwing darts.)

FOUNTAINHEAD

Slime Mold Algorithm





FOUNTAINHEAD

Data + Algorithms

FOUNTAINHEAD

Data + Algorithms

What does the future hold for Big Data algorithms? Discussion:

- More data + simple algorithms?
- Pipelines, Unix style. Stream data through small algorithms. (Infinite data stream?)
- Less data + sophisticated algorithms?
- Probably Approximately Correct (PAC). Ecorithms. ML.

Note: Parallel algorithms are a given!

FOUNTAINHEAD

Probably Approximately Correct (PAC)

In outline:

- Learner receives samples and must select a generalization function (called the hypothesis) from a certain class of possible functions.
- The goal is that, with high probability (the "probably" part), the selected function will have low generalization error (the "approximately correct" part).