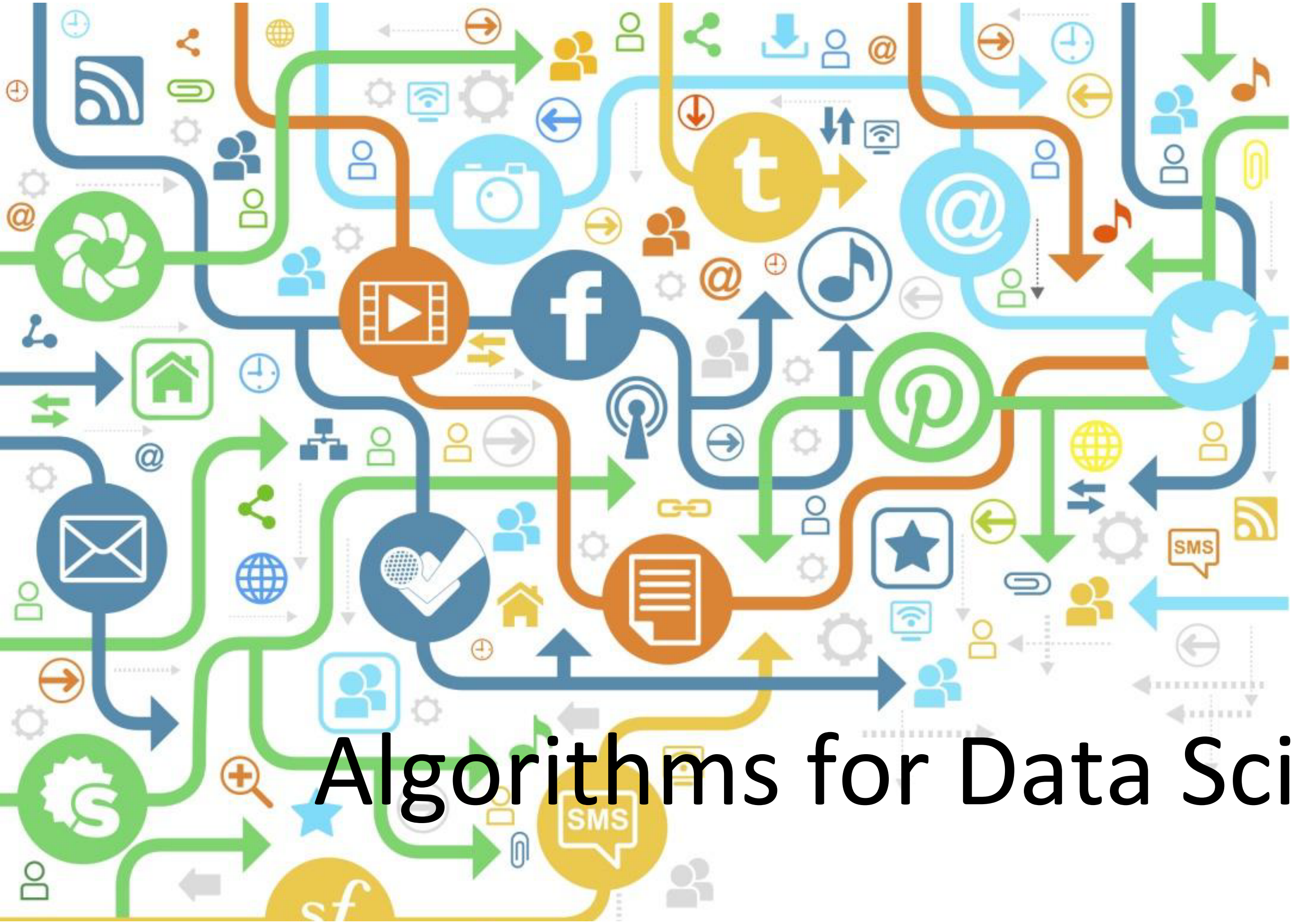
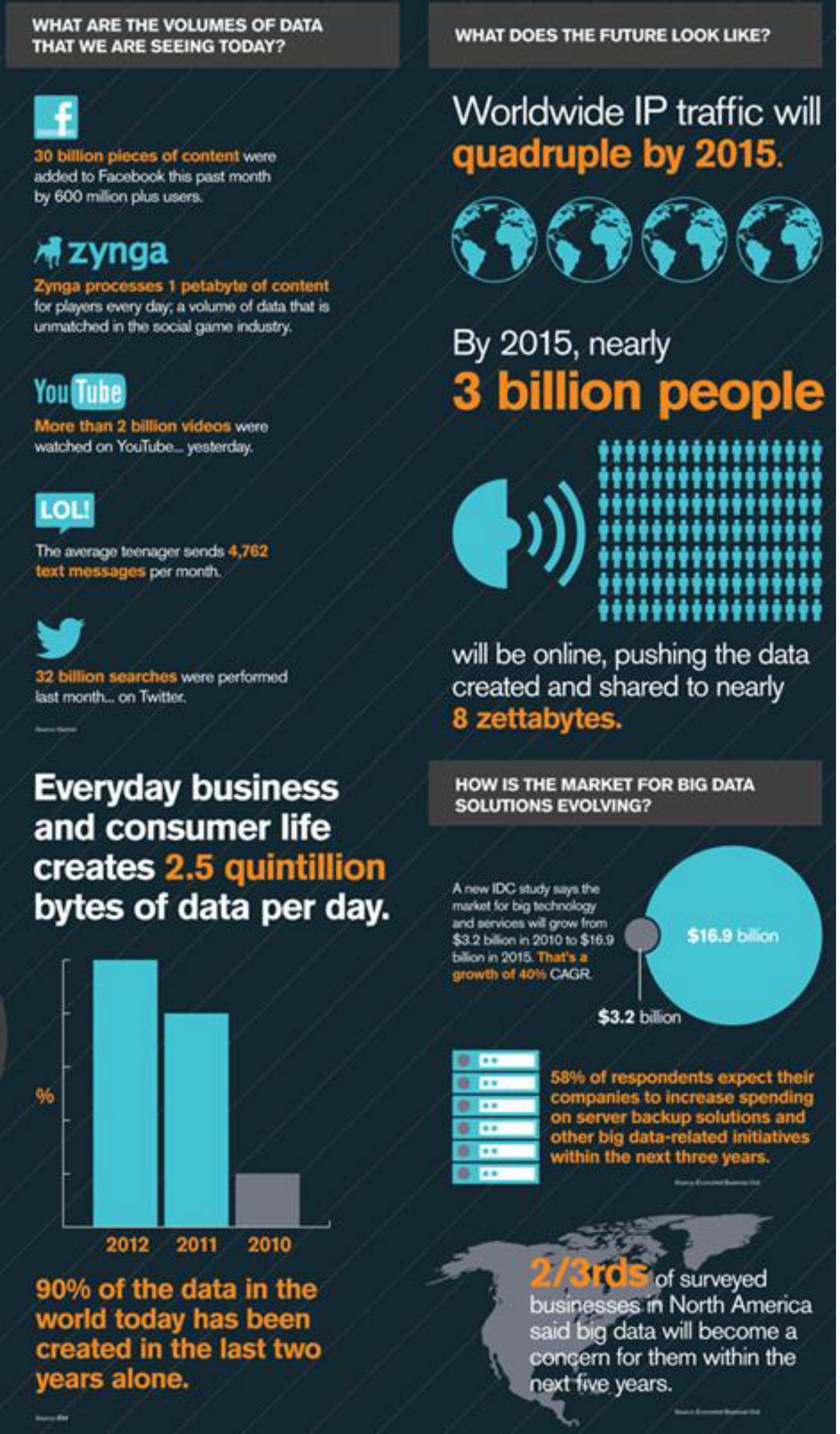


Algorithms for Data Science



Challenges of Big Data

- VOLUME
  - —Large amount of data
- VELOCITY
  - —Needs to be analyzed quickly
- VARIETY
  - —Different types of structured and unstructured data
- VERACITY
  - —Low quality data, inconsistencies

This Course

- Develop algorithms to deal with such data
  - Emphasis on different models for processing data
  - Common techniques and fundamental paradigms
  - Major applications where these techniques are useful
- Style: Algorithmic/ Theoretical
  - Background in basic algorithms (311) and probability (240) strictly required.

Grading

- Homeworks (4-5) in a group of 4
  - Will consist of mathematical problem/programming assignments
  - No late homework is allowed unless there are compelling reasons and preapproved by the instructor.
  - 20%
- Paper presentation
  - Each group will give a half an hour presentation on a paper selected by discussion with the instructor.
  - 30%
- Final Exam
  - One exam towards the end of the class
  - 50%

Office Hours

- Instructor: Thur 4-5pm at CS 322

- Teaching Assistant: My Phan
- ????

- All class related discussions should be done through piazza.

Tentative Syllabus

- Models
  - Developing FAST algorithms
  - Developing SMALL SPACE algorithms
  - Developing DISTRIBUTED algorithms
  - Developing algorithms through CROWD SOURCING
- Applications
  - Clustering
  - Estimating Statistical Properties
  - Algorithms over Massive Graphs and Social Networks
  - Machine Learning

Books

- Text Book: We will use reference materials from the following books. **Both can be downloaded for free.**
- Mining of Massive Datasets, Jure Leskovec, Anand Rajaraman and Jeff Ullman.
- Foundations of Data Science, a book in preparation, by John Hopcroft and Ravi Kannan

Models

- Different models need different algorithms for the same problem
  - Default: Main Memory Model
  - External Memory Model
  - Streaming Model
  - MapReduce
  - Crowdsourcing

1. Do you have enough main memory ?

2. How much disk I/O are you performing ?

3. Is your data changing fast ?

4. Can you distribute your data to multiple servers for fast processing ?

5. Is your data ambiguous that it needs human power to process ?

Counting Distinct Elements

Given a sequence A= $a_1, a_2, \ldots, a_m$ where $a_i \in \{1\ldots n\}$, compute the number of distinct elements in A (denoted by |A|).

- Natural and popular statistics, eg.
  - Given the list of transactions, compute the number of different customers (i.e. credit card numbers)
  - What is the size of the web vocabulary ?

Example: 4 5 5 1 7 6 1 2 4 4 4 3 6 6

distinct elements=7

Counting Distinct Elements

- **Default model: Random Access Main Memory Model**
- Maintain an array of size n: B[1,…,n]—initially set to all "0"
- If item "i" arrives set B[i]=1
- Count the number of "1"s in B

Counting Distinct Elements

- **Default model: Random Access Memory Model**
- Maintain an array of size n: B[1,…,n]—initially set to all "0"
- If item "i" arrives set B[i]=1
- Count the number of "1"s in B

> - O(m) running time ☺
> - Requires random access to B (?)
> - Requires space n even though the number of distinct elements is small or $m < n$ –domain may be much larger ✖

Counting Distinct Elements

- **Default model: Random Access Memory Model**
- Initialize count=0, an array of lists B[1….O(m)] and a hash function h : {1….n} → {1…O(m)}
- For each a_i
  - Compute j=h(a_i)
  - Check if a_i occurs in the list pointed to by B[j]
  - If not, count=count+1 and add a_i to the list
- Return count

Assuming that h(.) is random enough, running time is O(m), space usage O(m).

PROVE IT !

Space is still O(m)

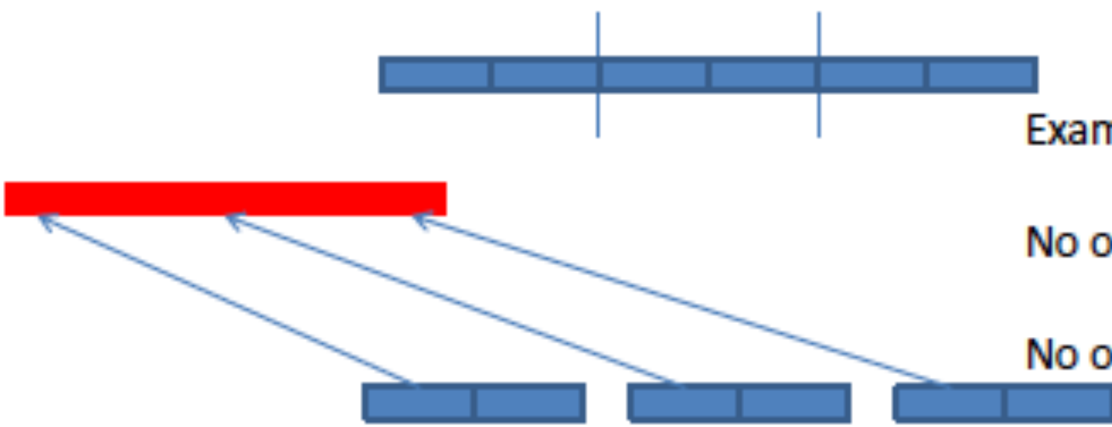
Random access to B for each input

Counting Distinct Elements

- **External Memory Model**
  - M units of main memory
  - Input size m, m >> M
  - Data is stored on disk:
    - Space divided into blocks, each of size B <=M
    - Transferring one block of data into the main memory takes unit time
  - Main memory operations for free but disk I/O is costly
  - Goal is to reduce number of disk I/O

Distinct Elements in External Memory

- Sorting in external memory
- External Merge sort
  - Split the data into M/B segments
  - Recursively sort each segment
  - Merge the segments using m/B block accesses



Example: M/B=3

No of disk I/O/merging= m/B

No of recursion call=$\log_{M/B} m$

Total sorting time=$m/B \log_{M/B} m$

Distinct Elements in External Memory

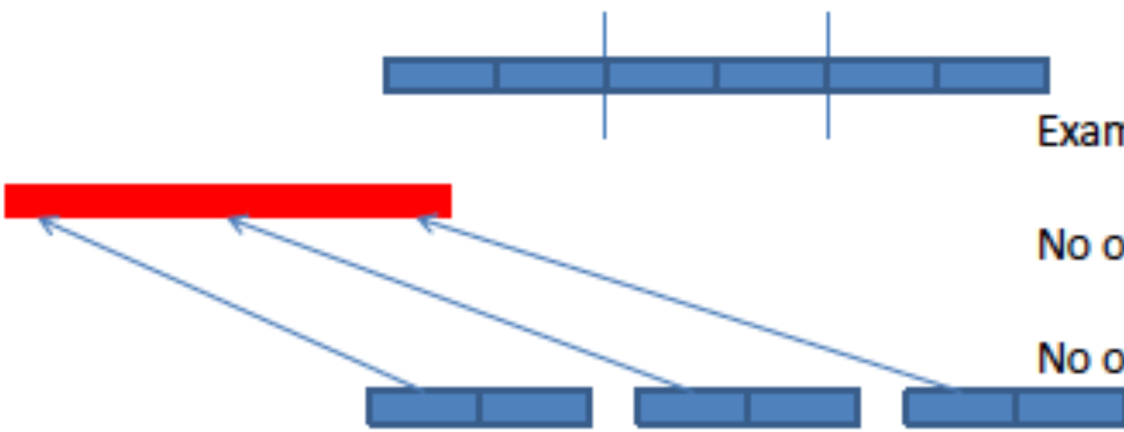
- Sorting in external memory
- External Merge sort
  - Split the data into M/B segments
  - Recursively sort each segment
  - Merge the segments using m/B block accesses

4 5 5 1 7 6 1 2 4 4 4 3 6 6

↓

1 1 2 3 4 4 4 4 5 5 6 6 6 7

Count=1
For j=2,…,m
If $a_j > a_{j-1}$ count=count+1



Example: M/B=3

No of disk I/O/merging= m/B

No of recursion call=$\log_{M/B} m$

Total sorting time=$m/B \log_{M/B} m$

Distinct Elements in Streaming Model

- Streaming Model
  - Data comes in streaming fashion one at a time

    (suppose from CD-ROM or cash-register)
  - M units of main memory, M << m
  - Only one pass over data
    - Data not stored is lost

Distinct Elements in Streaming Model

- Suppose you want to know if the number of distinct elements is at least "t"
- Initialize a hash function $h:\{1,\ldots n\} \rightarrow \{1,\ldots,t\}$
- Initialize the answer to NO
- For each a_i:
  - If $h(a_i) == 1$, then set the answer to YES

The algorithm uses only 1 bit of storage ! (not counting the random bits for h)

Distinct Elements in Streaming Model

- Suppose you want to know if the number of distinct elements is at least "t"
- Initialize a hash function h:{1,…m} → {1,…,t}
- Initialize the answer to NO, count=0
- For each a_i:
  - If $h(a_i)$ ==1, then count++ (this run returns YES)
- Repeat the above procedure for log n different hash functions from the family
  - Set YES if count > log n (1-1/e) [Boosting the confidence]

The algorithm uses log n bit of storage ! (not counting the random bits for h)

Run log(n) algorithms in parallel using t=2,4,8,…n
Approximate answers with high probability > 1-1/n
Space usage $O(\log^2 n)$

Distinct Elements in Streaming Model

- Suppose you want to know if the number of distinct elements is at least "t"
- Initialize a hash function h:{1,…m} → {1,…,t}
- Initialize the answer to NO, count=0
- For each a_i:
  - If $h(a_i)$ ==1, then count++ (this run returns YES)
- Repeat the above procedure for log n different hash functions from the family
  - Set YES if count > log n (1-1/e) [Boosting the confidence]

The algorithm uses log n bit of storage ! (not counting the random bits for h)

Run log(n) algorithms in parallel using t=2,4,8,…n
Approximate answers with high probability > 1-1/n
Space usage $O(\log^2 n)$

Approximation and Randomization are essential !

MapReduce Model

- Hardware is relatively cheap
- Plenty of parallel algorithms designed but
  - Parallel programming is hard
    - Threaded programs are difficult to test, debug, synchronization issues, more machines mean more breakdown
- MapReduce makes parallel programming easy

MapReduce Model

- **MapReduce makes parallel programming easy**
  - Tracks the jobs and restarts if needed
  - Takes care of data distribution and synchronization
- But there is no free lunch:
  - Imposes a structure on the data
  - Only allows for certain kind of parallelism

MapReduce Model

- Data:
  - Represented as <Key, Value> pairs
- Map:
  - Data → List < Key, Value> [programmer specified]
- Shuffle:
  - Aggregate all pairs with the same key [handled by system]
- Reduce:
  - <Key, List(Value)>→ <Key, List(Value)> [programmer specified]

Distinct Elements in MapReduce

- r servers
- Data
  - $[1,a_1], [2,a_2],\ldots, [n,a_m]$
- Map
  - $[1,a_1], [2,a_2],\ldots, [n,a_m] \rightarrow [1,a_1], [1,a_2],\ldots[1,a_{m/r}], [2,a_{m/r+1}],\ldots, [2,a_{2m/r}],\ldots,[r,a_m]$
- Reduce
  - Reducer 1: $[1,a_1], [1,a_2],\ldots[1,a_{m/r}] \rightarrow [1,a_1], [1,a_2],\ldots[1,a_{m/r}]$,[1,h()] generates the hash function)
  - Reducer 2: $[2,a_{m/r+1}], [2,a_{m/r+2}],\ldots[2,a_{2m/r}] \rightarrow [2,a_{m/r+1}], [2,a_{m/r+2}],\ldots[2,a_{2m/r}]$
  - ...
- Map

  $[1,a_1], [1,a_2],\ldots[1,a_{m/r}], [2,a_{m/r+1}],\ldots, [2,a_{2m/r}],\ldots,[r,a_m],[1,h()] \rightarrow [1,a_1], [1,a_2],\ldots[1,a_{m/r}], [2,a_{m/r+1}],\ldots, [2,a_{2m/r}],\ldots,[r,a_m]$, [1,h()], [2,h()], ..., [r,h()] makes multiple copies of the hash function for distribution
- Reduce
  - Reducer 1: $[1,a_1], [1,a_2],\ldots[1,a_{N/r}]$,[1,h()], create sketch B_1, outputs $[1,B_1]$
  - ......
- Map
  - $[1,B_1], [2,B_2],\ldots,[r,B_r] \rightarrow [1,B_1], [1,B_2],\ldots,[1,B_r]$ gathers all the sketches
- Reduce
  - Reducer1: $[1,B_1], [1,B_2],\ldots,[1,B_r]$, computes $B= B_1+B_2+\ldots+B_r$, Follows the Streaming Algorithm to compute distinct elements from the sketch

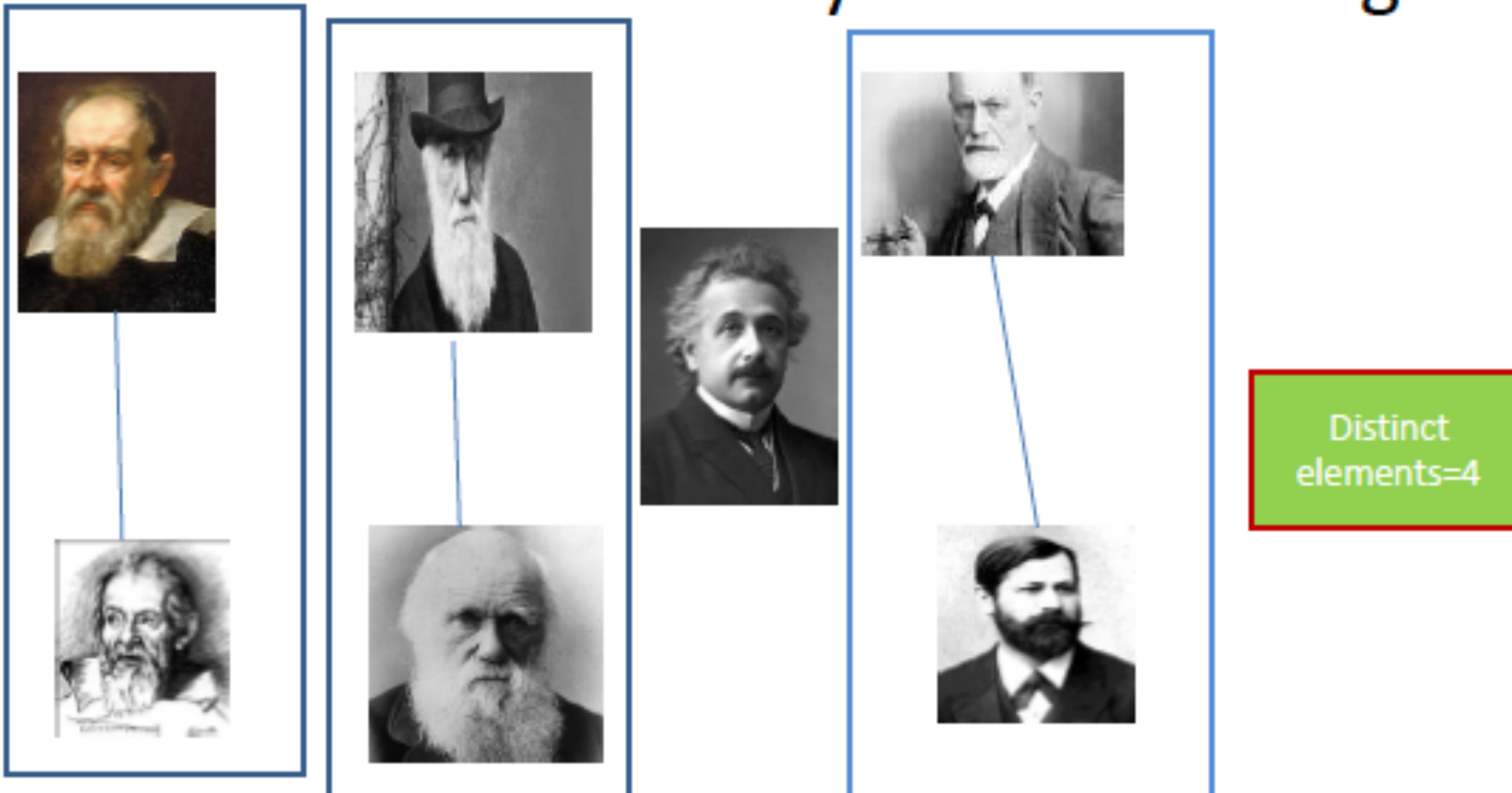
Crowdsourcing

- Incorporating human power for data gathering and computing
- People still outperform state-of-the-art algorithms for many data intensive tasks
  - Typically involve ambiguity, deep understanding of language or context or subjective reasoning

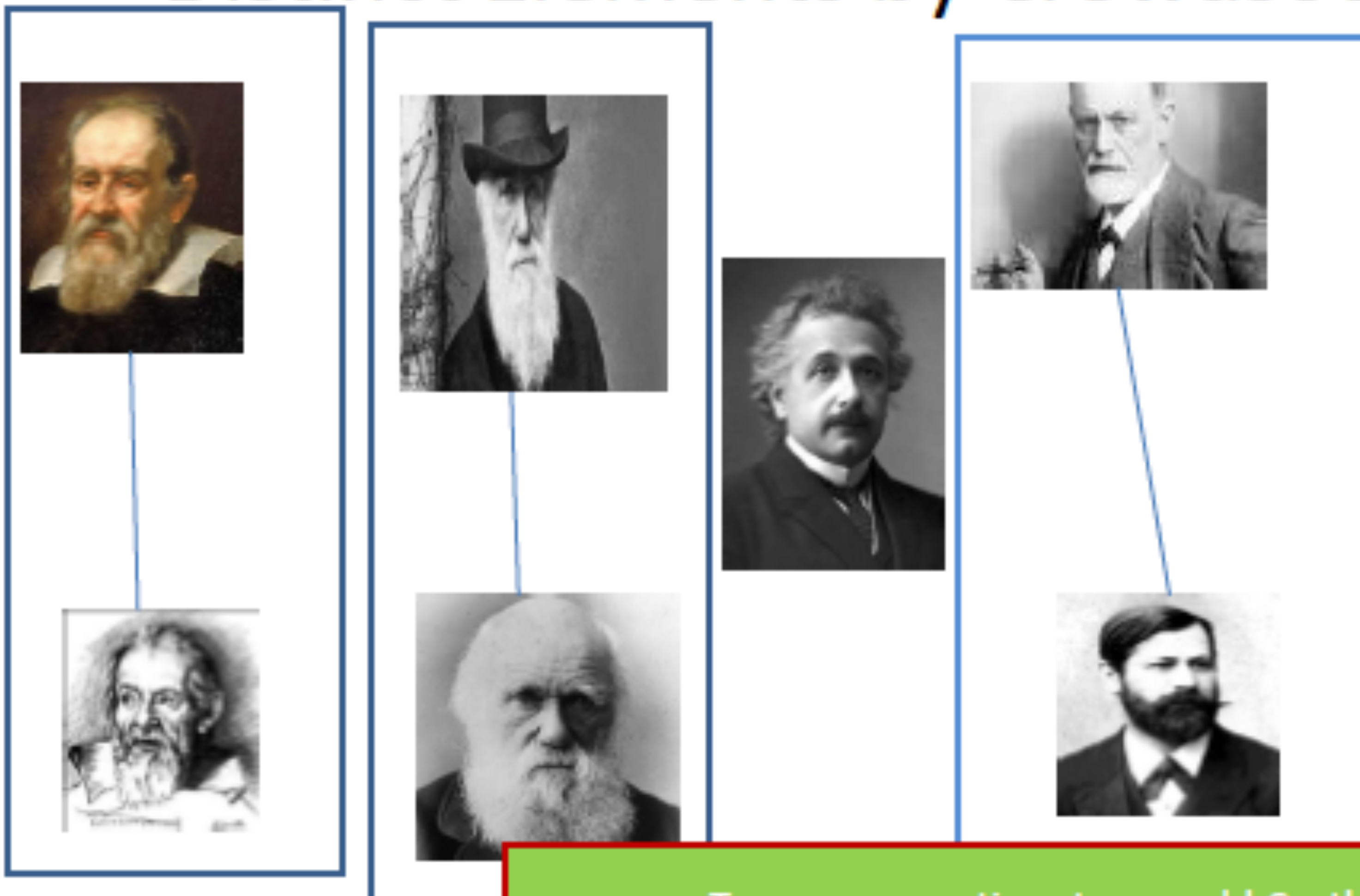
Distinct Elements by Crowdsourcing

- Ask for each pair if they are equal
- Create a graph with each element as node
- Add an edge between two nodes if the corresponding pairs are returned to be equal
- Return number of connected components
- Also known as record linkage, entity resolution, deduplication

Distinct Elements by Crowdsourcing



Distinct Elements by Crowdsourcing



Distinct elements=4

Too many questions to crowd ! Costly.
Can we reduce the number of questions ?