Combinatorial Group Testing

- Much of the current effort of the ***Human Genome Project*** involves the screening of large DNA libraries to isolate clones containing a particular DNA sequence(s).
- This screening is important for *disease-gene mapping* and also for *large-scale clone mapping*.
- Efficient screening techniques can facilitate a broad range of basic and applied biological research.

Combinatorial Group Testing

- When screening DNA libraries, several factors determine the cost of identifying desired objects, including:
  - ***1)*** the same library is screened with many different probes.
  - ***2)*** it is expensive to prepare a pool for testing the first time, although once a pool is prepared, it can be screened many times with different probes.
  - ***3)*** screening one pool at a time is expensive. Screening pools in parallel with the same probe is cheaper.
  - ***4***) there are constraints on pool sizes. If a pool contains too many different clones, then positive pools can become too dilute and could be mislabeled as negative pools.

Combinatorial Group Testing

- In ***non-adaptive group testing***, one must decide exactly which pools to test before any testing occurs.
- A non-adaptive group testing algorithm is sometimes referred to as a ***one-stage algorithm***. Every parallel algorithm is non-adaptive.
- An alternative ***two-stage algorithm*** is a nearly non-adaptive algorithm, in which, an initial batch of simultaneous tests is carried out, then using the information from the first stage, the second batch of simultaneous tests is computed and carried out.
- Because of factors 1, 2, and, 3 (see slide 2) *weakly-adaptive two-stage algorithms* are generally preferred when screening DNA libraries.

Combinatorial Group Testing

- *Combinatorial Group Testing* refers to the situation in which one is given:
  - A (very) large *set of objects* O, and
  - an unknown (relatively small) *subset* $P \subseteq O$.
- The *task* is to determine the content of P by asking minimal number of queries of the type *"does* P *intersect* Q*?"*, where Q is a subset of O.

Example of Group Testing

- Set of coins
  - O={  }

- Set $P \subseteq O$, *where* P={  }

- *Pool* $Q \subseteq O$; *Query* "does $P \cap Q \neq \varnothing$?"
  - E.g., for Q={1,3,6} the answer is *YES (1);*
  - And for Q={2,4,5} the answer is *NO (0);*

Non-Adaptive CGT

- ***Definition:*** A *d-disjunct* matrix, a.k.a, *(d,d,n)-selector* is defined as any *n*-column binary matrix *M*, such that:
  - For any *d* columns $c_1, c_2, \ldots, c_d$ in *M* there exist *d* rows $r_1, r_2, \ldots, r_d$, s.t., entries in *M* available on *intersection* of selected *d columns* and *rows* form a *permutation matrix.* I.e., it contains all different unit vectors.

	c_1	c_2	.....	c_d
	----------	----------	----------	----------
r_1	---------1---------	---------0---------	---------0---------	----------
	----------	----------	----------	----------
r_2	---------0---------	---------0---------	---------1---------	----------
	----------	----------	----------	----------
	----------	----------	----------	----------
r_d	---------0---------	---------1---------	---------0---------	----------
	----------	----------	----------	----------

2-disjunct matrix - example

- *2-disjunct* matrix for *n =8* items based on *binary representation* of numbers *0,1,…,7*

0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1
1	1	1	1	0	0	0	0
1	1	0	0	1	1	0	0
1	0	1	0	1	0	1	0

- *2-disjunct (d-disjunct) matrix can be used to determine* $P \subseteq O$ *of cardinality 1 (d-1)*

Feedback Vector

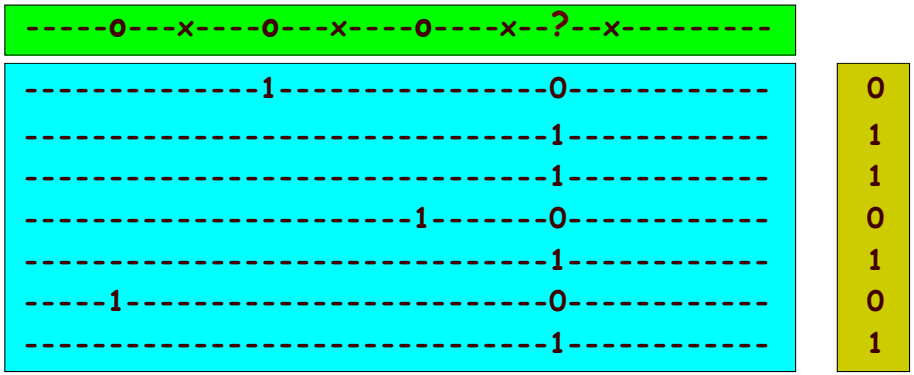
Population vector, where *x* stands for elements of *P*

---------x---------x---------x-----x---------	
---------0---------0---------0-----0---------	0
---------1---------0---------1-----0---------	1
---------0---------1---------0-----0---------	1
---------0---------0---------0-----0---------	0
---------0---------0---------0-----1---------	1
---------0---------0---------0-----0---------	0
---------1---------1---------0-----1---------	1

Feedback vector

Evidence against *P* membership

There is an evidence against membership in *P* for **o** items



Non-adaptive Group Testing

- The size of the *d-disjunct* matrix
  - Lower bound $\Omega(d^2 \log n/\log d)$
  - Upper bound $O(d^2 \log (n/d))$
    - [*Dyachkov, Rykov & Rashad ('82, '89)*]
- ***Theorem:*** The Combinatorial Group Testing problem, with $|P|=d-1$, can be solved in one stage and $O(d^2 \log (n/d))$ queries/tests.

Fully Adaptive Group Testing

- *In contrast* there exists *fully adaptive* Combinatorial Group Testing algorithm that uses $O(d \log (n/d))$ tests (*as well as stages*)
- The upper bound $O(d \log (n/d))$ matches asymptotically the *information theory lower bound* for Combinatorial Group Testing with d unknown items, which is $\Omega(\log \binom{n}{d})$
- *Can we achieve this bound in 2 stages?*

(d,m,n)-selectors

- ***Definition:*** *(d,m,n)-selectors* is defined as any n-column binary matrix M, such that:
  - For any columns $c_1, c_2, \ldots, c_d$ in M there exist m rows $r_1, r_2, \ldots, r_m$, s.t., entries in M available on *intersection* of selected d *columns* and m *rows* form m different rows of permutation matrix of size $d \times d$.
- One can prove that there exist *(d,m,n)-selectors* of size (number of rows) $O(d^2 \cdot \log(n/d)/(d-m+1))$
  - Recall that *(d,d,n)-selectors* correspond to *d-disjunct matrices*
  - However, do far there is not known efficient deterministic construction of *(d,m,n)-selectors!*

Weakly adaptive 2-stage algorithm

- **Stage 1**: Apply $(2d,d+1,n)$-selector on population with $|P| < d$
  - Compute feedback vector
  - Generate evidence against membership in P
  - The number of the items without the evidence is $< 2d$
- **Stage 2:** Check the items without the evidence in $< 2d$ separate pools
- Theorem: There is a 2-stage group testing algorithm for that works in time $O(d \cdot \log(n/d))$.

Weakly adaptive 2-stage algorithm

- Proof of the ***Theorem:***
  - In general the size of (d,m,n)-selector is $O(d^2 \cdot \log(n/d)/(d-m+1))$ and in particular the size of $(2d,d+1,n)$-selector is $O((2d)^2 \cdot \log(n/2d)/(2d-(d+1)+1)) = O(d \cdot \log(n/d))$.
  - *Proof by contradiction*. Assume that the number of items without the evidence is $\geq 2d$. And consider any $2d$ items without the evidence and their respective $2d$ columns in $(2d,d+1,n)$-selector. At least $d+1$ items (among $2d$) will be separated from each other in the $(2d,d+1,n)$-selector, so even if d out of $d+1$ belong to P there is at least one item that should've gathered the evidence against membership in P. Which contradicts the assumption.

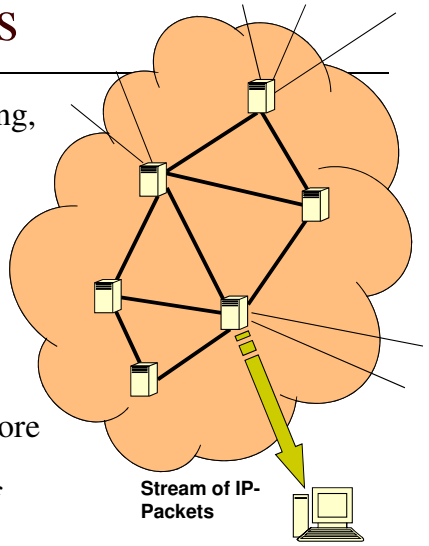
Streaming data sources

- Internet traffic monitoring
- Web logs and click streams
- Financial and stock market data
- Retail and credit card transactions
- Telecom calling records
- Sensor networks, surveillance
- RFID
- Instruction profiling in microprocessors
- Data warehouses (random access too expensive).

Internet Traffic Analysis

- Usage trends for engineering, provisioning, abuse detection, etc.
- ***Discover sources of large traffic***

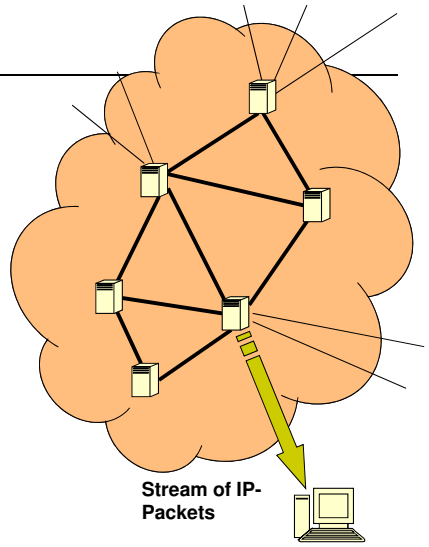
- Items = IP packets
- ID = Flow ID
  - E.g., sender's IP address
- Frequent items = ***Heavy hitters***
  - E.g., report all flows that consume more than 1% of the link bandwidth.
  - Counting bytes, instead of number of occurrence.



Stream Data

- Rapid, continuous arrival:
  - Several million packets/sec
- Huge volume:
  - > 50 TB of header data per day for Gigabit router
- Real time response
- Small memory: fast but costly SRAM

- ***In the sea of data, spot unusual traffic patterns and anomalies***



Streaming model

- Motivating Scenarios
  - Data flowing directly from generating source
  - "Infinite" stream cannot be stored
  - Real-time requirements for analysis
- Model
  - Stream – at each step can request next input value
  - Assume stream size n is finite/known (fix later)
  - Memory size M << n, possibly O(1) size

Finding majority in streaming model

- Given a sequence of streamed *n* items and a constant size memory.
- In a single pass, decide if some item in the stream is in clear majority (occurs >*n*/2 times)?

2	9	9	9	7	6	4	9	9	9	3	9

n = 12, *where item* 9 *is in clear majority*

Misra-Gries Algorithm

- Adopt a single counter *count* = 0 and associated ID, and iterate
  - **if** (count>0) then
    if (new item = stored ID) *count*++;
    then count ++
    else count --;
  else store the new item with count = 1.
- In the end, if counter > 0, associated ID links to the only candidate.

	2	9	9	9	7	6	4	9	9	9	3	9
ID	2	2	9	9	9	9	4	4	9	9	9	9
count	1	0	1	2	1	0	1	0	1	2	1	2

A generalization: frequent items

Finding k items, each occurring at least $n/(k+1)$ times.

ID	ID_1	ID_2	.	.	.	.	ID_k
count	.	.					

- Sketch of the algorithm:
  - maintain k items, and their associated counters;
  - if next item x is one of the k, increment respective counter;
  - else if a zero counter available, associate x with it and set count = 1;
  - else (all counters non-zero) decrement **all** k counters

Frequent Elements: Analysis

- A frequent item's counter is decremented if all counters are full*: it erases k+1 items.*
- If *x occurs > n/(k+1)* times, then it cannot be completely erased.
- Similarly, x must get inserted at some point, because there are not enough items to keep it away.

Problem of False Positives

- ***False positives*** in Misra-Gries algorithm
  - It identifies all true heavy hitters, but not all reported items are necessarily heavy hitters.
  - How can we tell if the non-zero counters correspond to true heavy hitters or not?
- A second pass is needed to verify.
- False positives are problematic if heavy hitters are used for billing or punishment.
- What guarantees can we achieve in one pass?

Approximation Guarantees

- Find heavy hitters with a guaranteed approximation error
- E.g., Manku-Motwani (*Lossy Counting*)
  - Suppose you want ***φ-heavy*** hitters, i.e., items with freq > φN
  - An approximation parameter ε, where ε << φ.
    (E.g., φ = .01 and ε = .0001; φ = 1% and ε = .01%)
  - ***Identify all items with frequency > φN***
  - ***No reported item has frequency < (φ - ε)N***
- The algorithm uses O(1/ε log (εN)) memory