Detour: Bloom Filters

Say we want to store a set $S \subseteq [n]$, $|S| = m \ll n$.

Recall: hash tables let us do this using $O(m \log n)$ bits of space, and, depending on the implementation, supported insert, delete, and lookup operations in expected or worst-case constant time with high probability.

(Also, note that these hash tables allowed enumeration of all items in S in time $O(m)$, and also allowed associating keys with values).

We will now see a method for <u>approximately</u> representing S using just $O(m)$ bits. This method supports only lookup and insert operations, and does not support associating keys and values.

Bloom Filter:

- Let k be a constant (say 5).
- Maintain M bits, initialized to 0. View the bits as k tables, each consisting of $\frac{M}{k}$ bits. $\swarrow T_1 \ldots T_k$
- Let $h_1, \ldots h_k : [n] \to [M/k]$ be random hash functions.
- Insert(x):

```
    For $i = 1 \ldots k$
        $T_i[h_i(x)] \leftarrow 1$
```

- Lookup(x):

```
    If $T_i[h_i(x)] = 1$ for all $i \in [k]$, output 1. Else output 0.
```

Inserts and lookups require just k hashes and k bits read.

Claim 1: If $x \in S$, then Lookup(x) returns 1 with probability 1. "No false negatives"

Proof: Obvious from definition of Insert and Lookup operations.

Claim 2: If $x \notin S$, then $\Pr_{h_1, \ldots, h_k}[\text{Lookup}(x) \text{ returns } 1] = \left(\left(1-\frac{k}{m}\right)^{m}\right)^{k}$

$$\approx \left(1 - e^{-mk/m}\right)^{k}.$$

"False positives are rare"

Example: Consider a set S of 100,000 common passwords, which are 7 characters long on average. This requires 700,000 bytes to store exactly. They might be compressible to 300,000 bytes or so, but then inserts and lookups might take a long time.

Instead, keep a 100,000-byte Bloom Filter, consisting of $k = 5$ tables, each with 160,000 bits. Then false positive probability is ~2% and inserts and lookups take 5 hash evaluations

Sparse Recovery in Data Streams via IBLTs

Consider a turnstile stream $\sigma = \langle (a_1, \delta_1), \ldots (a_m, \delta_m) \rangle$ under the promise that at the end of the stream, $f_i = 0$ for all but at most N items i.

E.g. Flows through a router.

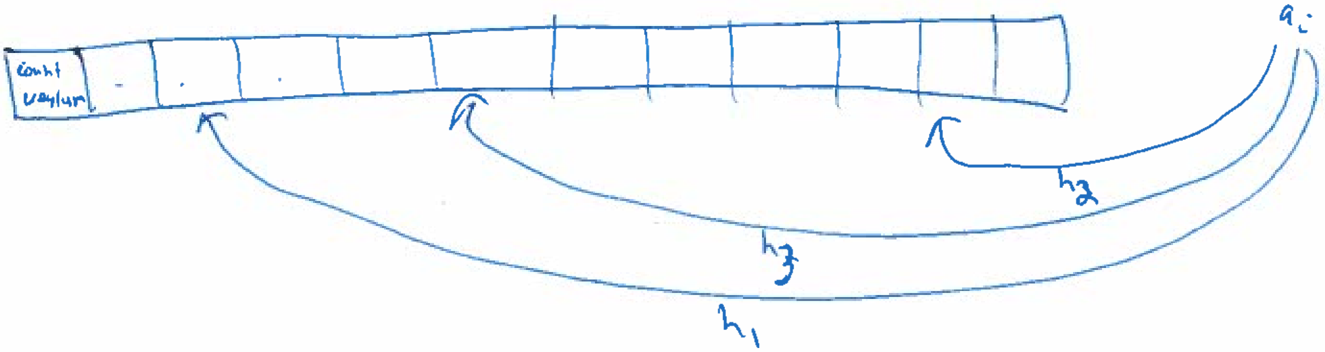
- Goal: List all items with non-zero frequency, using space $O(N \log n)$ in bits.
- We will use this next lecture to build algorithms for more complicated problems.

For simplicity, assume that at end of stream no item has frequency more than 1.

IBLT algorithm:

- Maintain M cells, where each cell stores a count and a keysum all initialized to 0
- Let $h_1, \ldots h_r : [n] \to M$ be random hash functions
- While processing update (a_i, δ_i):

```
For j=1..r:
    • Add $\delta_i \cdot a_i$ to keysum field of cell $h_j(a_i)$
    • Add $\delta_i$ to count for cell $h_j(a_i)$
```



Listing Algorithm:

- Call a cell "pure" if exactly one item with non-zero frequency hashes to it.
  - Under assumption that all items have frequency ≤ 1, can check if a cell is pure just by looking whether count $= 1$. If so, can identify the item hashing to it by looking at key sum field.
- While there exists a pure cell:
  - Output the unique item $i: f_i > 0$ hashing to the cell.
  - Let $\delta =$ count of pure cell (assumed to be 1 by)
  - Call Insert$(i, -\delta)$.

- In case where frequencies may be > 1, can still identify pure cells and the unique item hashing to it (with high probability, using fingerprints).
  - Details later. (Actually, on problem set #3).

Key Question: How many cells are required to ensure the listing algorithm successfully recovers all items with non-zero frequency?

Peeling Algorithms and Their Analysis

- A hypergraph is just like a standard graph (vertices and edges connecting them) except edges may contain more than 2 vertices.
  - A hypergraph is r-uniform if each edge contains r-vertices.

- An IBLT naturally defines a corresponding hypergraph G
  - IBLT cells $\Leftrightarrow$ vertices of G
  - Items in IBLT $\Leftrightarrow$ hyperedges of G
  - G is r-uniform if IBLT uses r hash functions

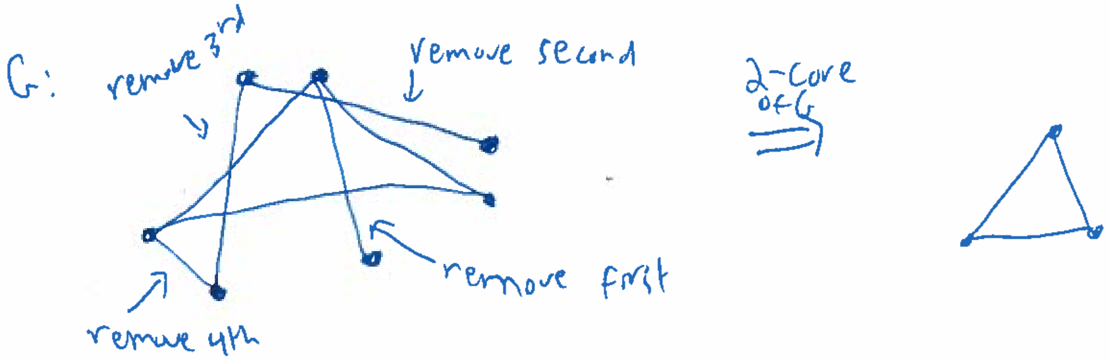
Note: If IBLT uses truly random hash functions, then G is a random r-uniform hypergraph with M nodes and N edges.
↑ # cells in IBLT
↖ # of $i: f_i > 0$ in IBLT

e.g.

The Listing Algorithm for the IBLT corresponds to running the following algorithm on G.

- While there exists a vertex v in G of degree < 2
  - Remove v and all of its incident edges

- The remaining graph when the above algorithm gets stuck is called the 2-core of G.
- Listing succeeds when 2-core is empty.
- Example in case of a simple graph:



- In simple graphs G, the 2-core is empty iff G does not contain a cycle.
  - In hypergraphs, the 2-core is a more complicated object.

- Consider a random r-uniform hypergraph G with M nodes and $N = c\cdot M$ edges,
  - i.e., each edge has r vertices, chosen uniformly at random from $[M]$

- Known Fact: Appearance of a non-empty k-core obeys a sharp threshold.
  - For some constant $c_{k,r}$, when $N < c_{k,r}\cdot M$ by a constant factor, the k-core is empty with probability $1-o(1)$.
  - When $N > c_{k,r}\cdot M$ by a constant factor, the k-core is non-empty with probability $1-o(1)$.

  - Implication: to successfully list a set of size N with probability $1-o(1)$, the IBLT needs $\frac{N}{c_{k,r}}$ cells.

  - E.g., $c_{2,3} \approx 0.818$, $c_{2,4} \approx 0.772$, $c_{3,3} \approx 1.553$

  - In general, $c_{k,r} := \min\limits_{x>0} \dfrac{x}{r\cdot\left(1-e^{-x}\cdot\sum_{j=0}^{k-2}\frac{x^j}{j!}\right)^{r-1}}$.