Data Structures and Algorithms
Cardinality

CS 225
Brad Solomon

December 1, 2025

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

(Potentially) double exam week!

Hope you had a good Fall break!

Exam 5 is 12/3 — 12/5

Retake exam is 12/7 — 12/9

Remember retake is OPTIONAL and can lower your grade!

During exam block you will have exams 0 — 4 as options

ONLY OPEN ONE EXAM IN THE LIST!

Learning Objectives

Review bloom filters and identify the ‘weakness’ of BFs

Introduce the concept of cardinality and cardinality estimation

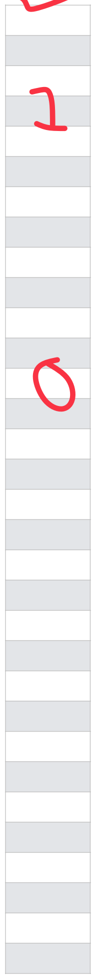
Bloom Filters

A probabilistic data structure storing a set of values

$h_{\{1,2,3,\ldots,k\}}$

$h(x) \rightarrow 1$

$h(y) \rightarrow 0$



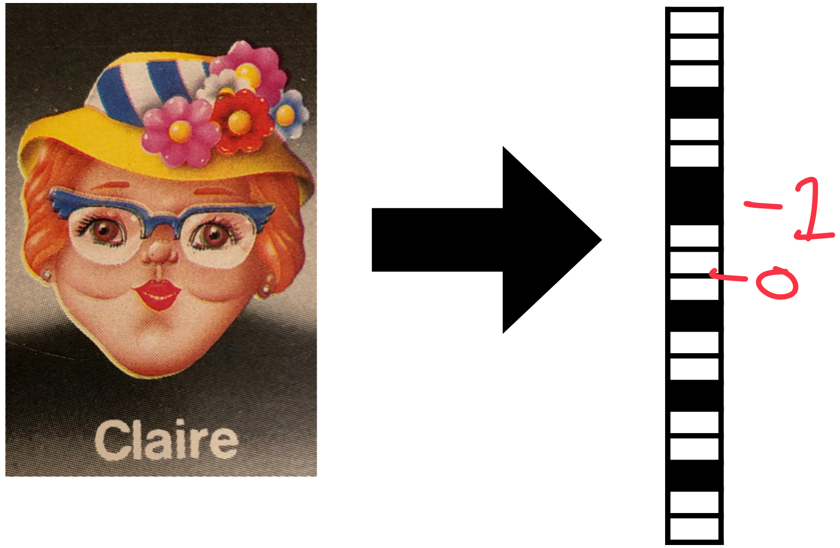
Has three key properties:

- k, number of hash functions
- n, expected number of insertions
- m, filter size in bits

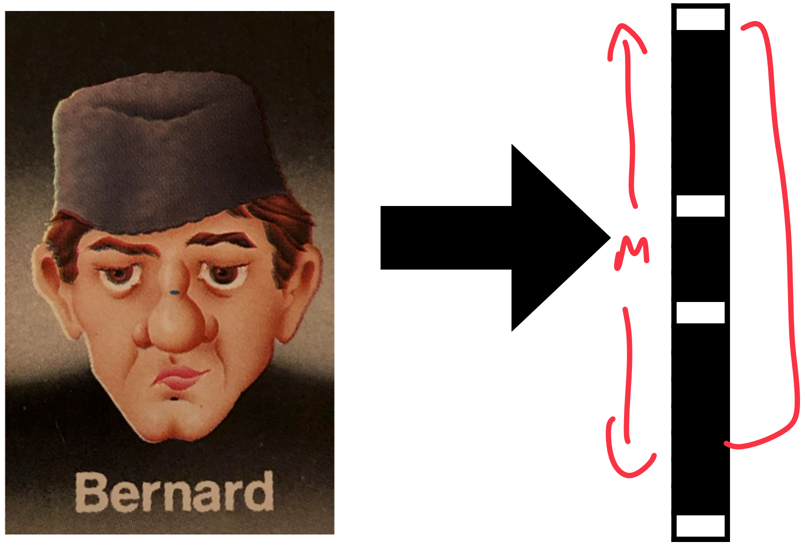
Expected false positive rate: $$\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^{k} \approx \left(1-e^{\frac{-nk}{m}}\right)^{k}$$

Optimal accuracy when: $$k^* = \ln 2 \cdot \frac{m}{n}$$

The hidden problem with (most) sketches…



1 million unique sequences



4 million unique sequences

my filter is saturated!

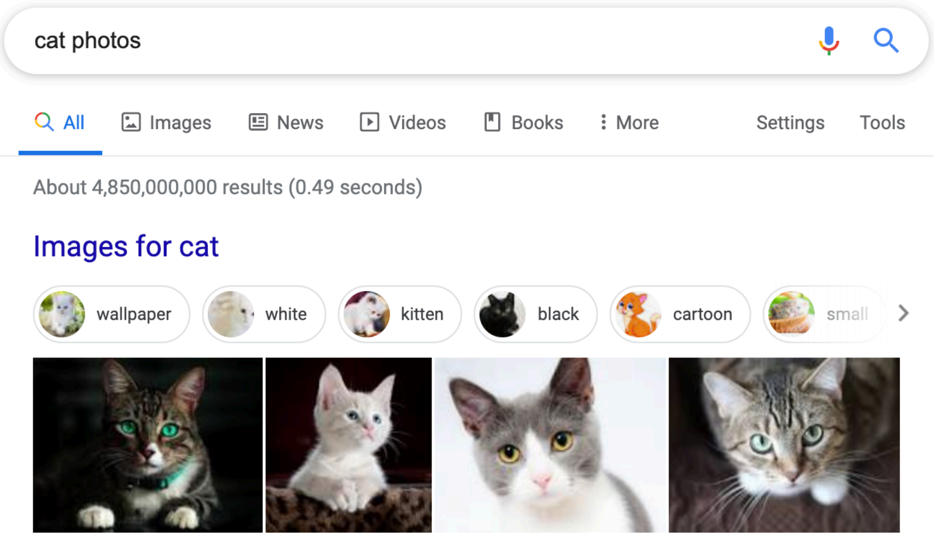
Cardinality

Cardinality is a measure of how many unique items are in a set

2
4
9
3
7
9
7
8
5
6

Cardinality

Sometimes its not possible or realistic to count all objects!



Estimate: 60 billion — 130 trillion

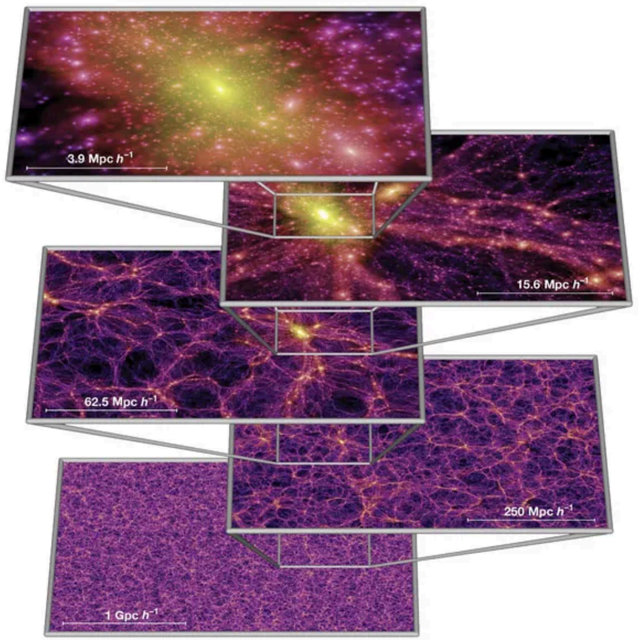


Image: https://doi.org/10.1038/nature03597

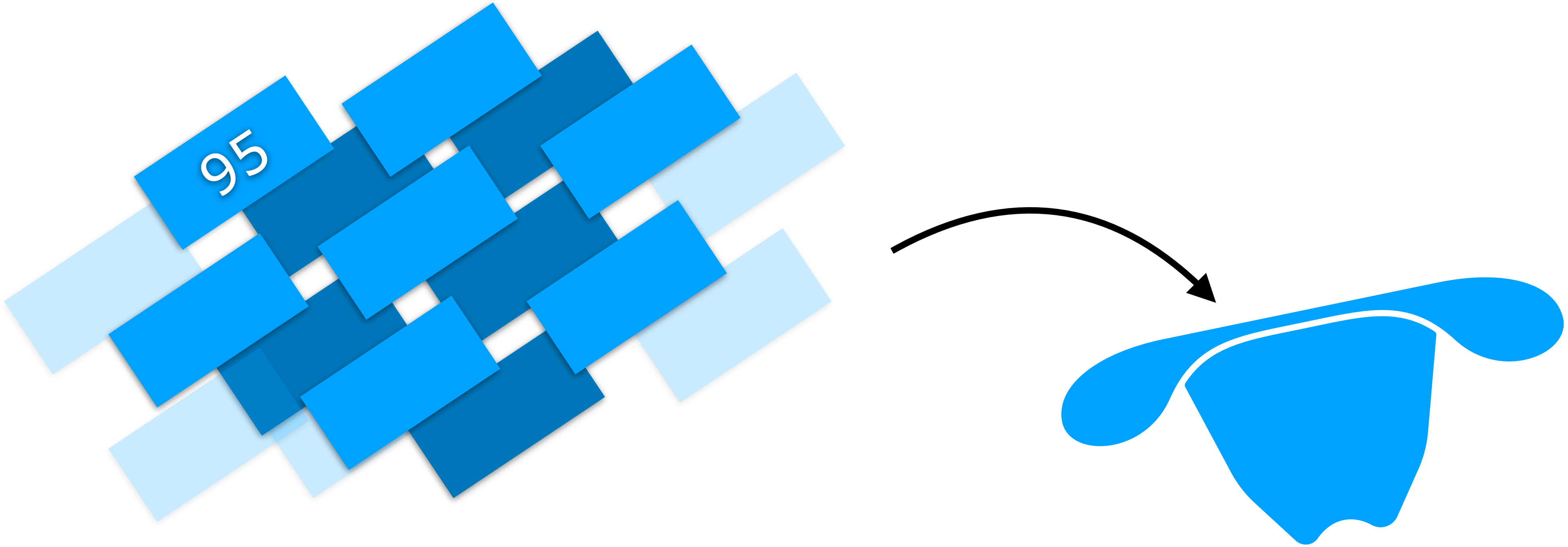
5581
8945
6145
8126
3887
8925
1246
8324
4549
9100
5598
8499
8970
3921
8575
4859
4960
42
6901
4336
9228
3317
399
6925
2660
2314

Cardinality Estimation

Imagine I fill a hat with numbered cards and draw one card out at random.

If I told you the value of the card was 95, what have we learned?

↳ just that 95 exists



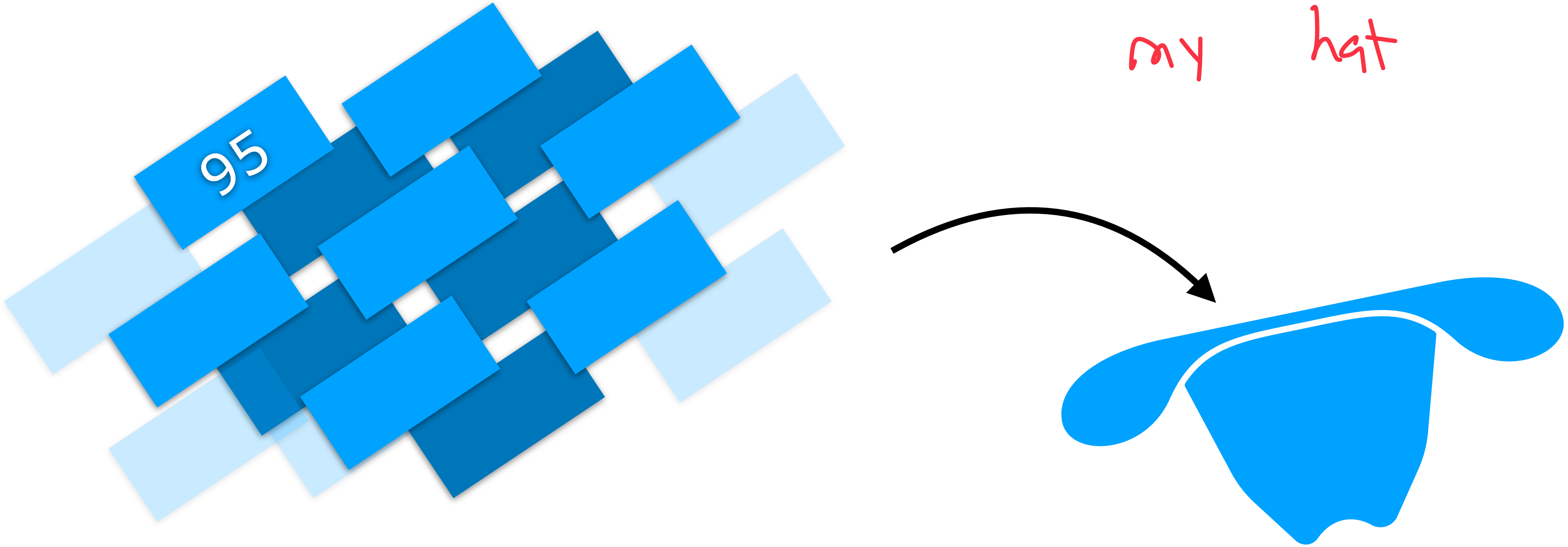
Analogy from Ben Langmead

Cardinality Estimation

Imagine I fill a hat with **a random subset** of numbered cards **from 0 to 999**

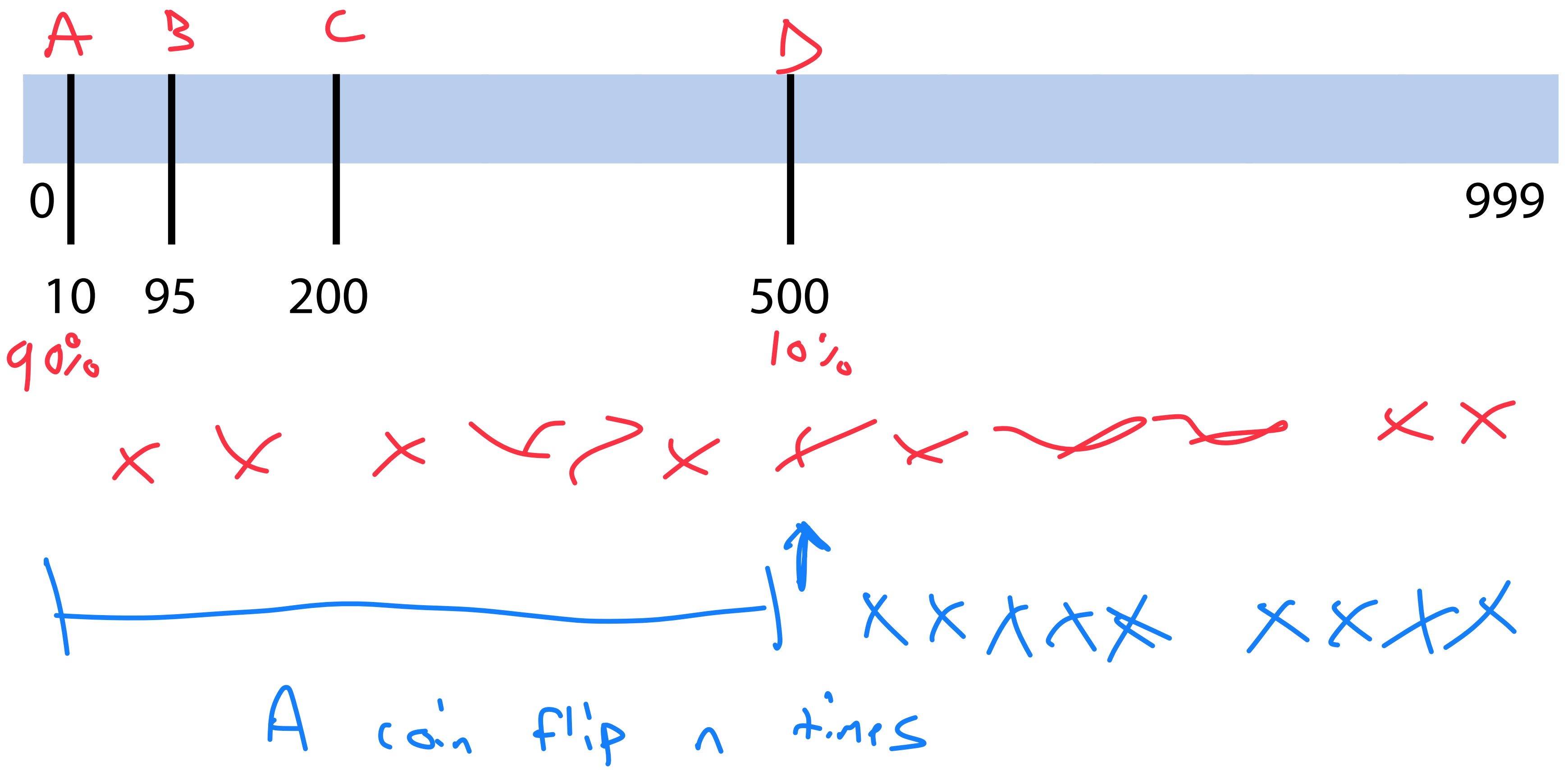
If I told you that the **minimum** value was 95, what have we learned?

↳ there are 10 items in my hat



Cardinality Estimation

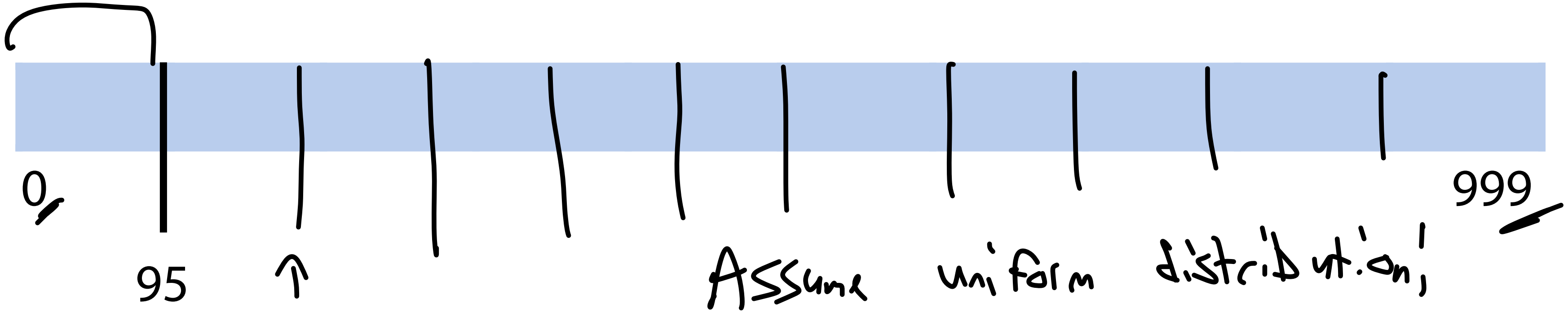
Imagine we have multiple uniform random sets with different minima.



Cardinality Estimation

Let min = 95. Can we estimate N, the cardinality of the set?

If this is range for one item



This is 2

Assume uniform distribution!

$$95 \approx \frac{1000}{N+1} \rightarrow N \sim 9.5 = 10$$

Cardinality Estimation

Let min = 95. Can we estimate N, the cardinality of the set?

Claim: $95 \approx \dfrac{1000}{(N+1)}$

Cardinality Estimation

Let min = 95. Can we estimate N, the cardinality of the set?

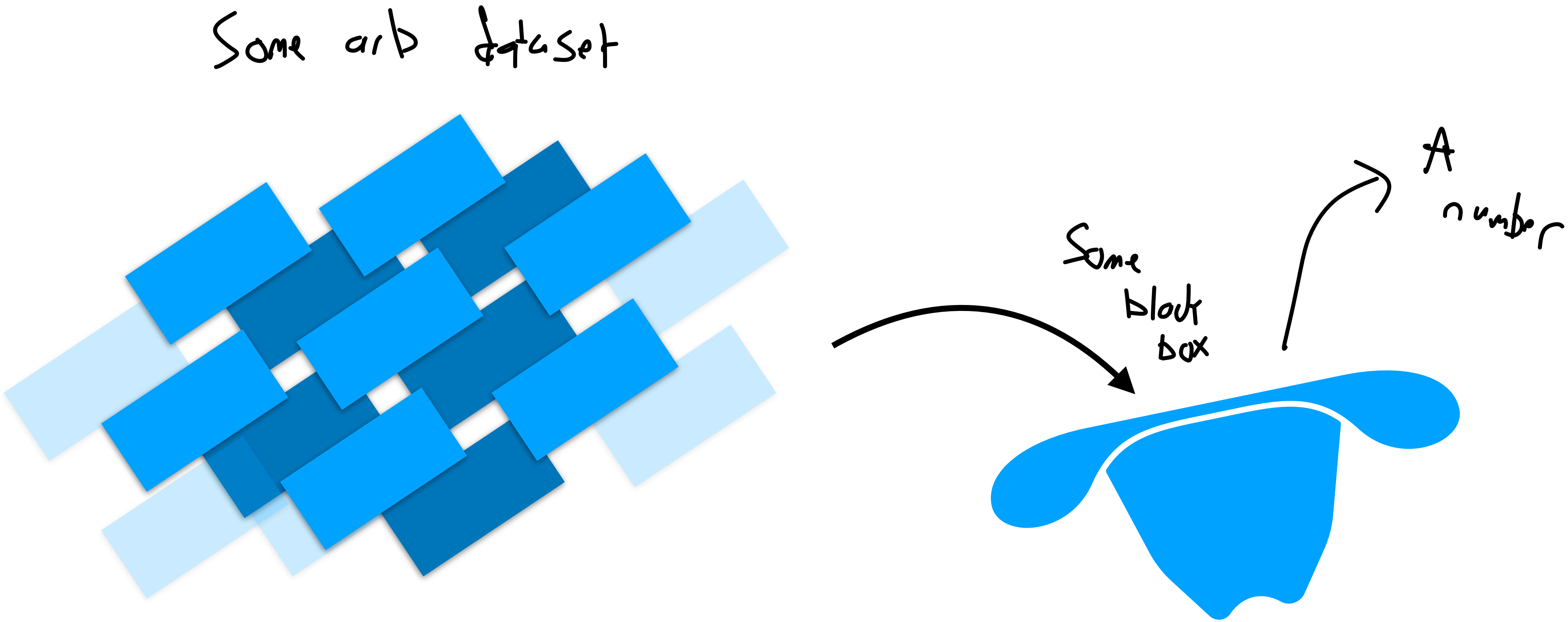
Conceptually: If we scatter N points randomly across the interval, we end up with $N + 1$ partitions, each about $1000/(N+1)$ long

Assuming our first 'partition' is about average:

$$95 \approx 1000/(N+1)$$
$$N+1 \approx 10.5$$
$$N \approx 9.5$$

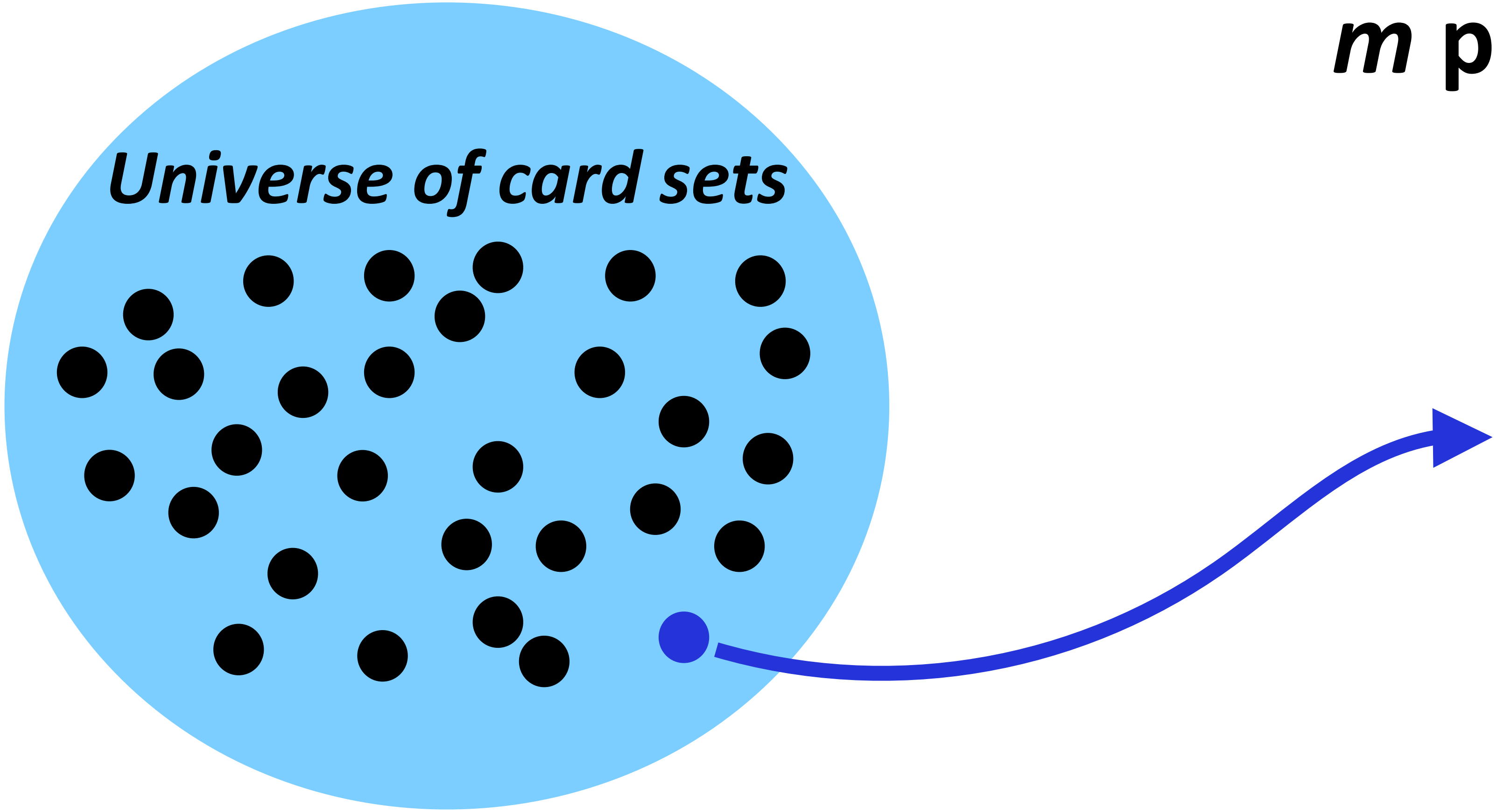
Cardinality Estimation

Why do we care about "the hat problem"?



Cardinality Estimation

Why do we care about “the hat problem”?



m **possible minima**

Key	Value

Cardinality Estimation

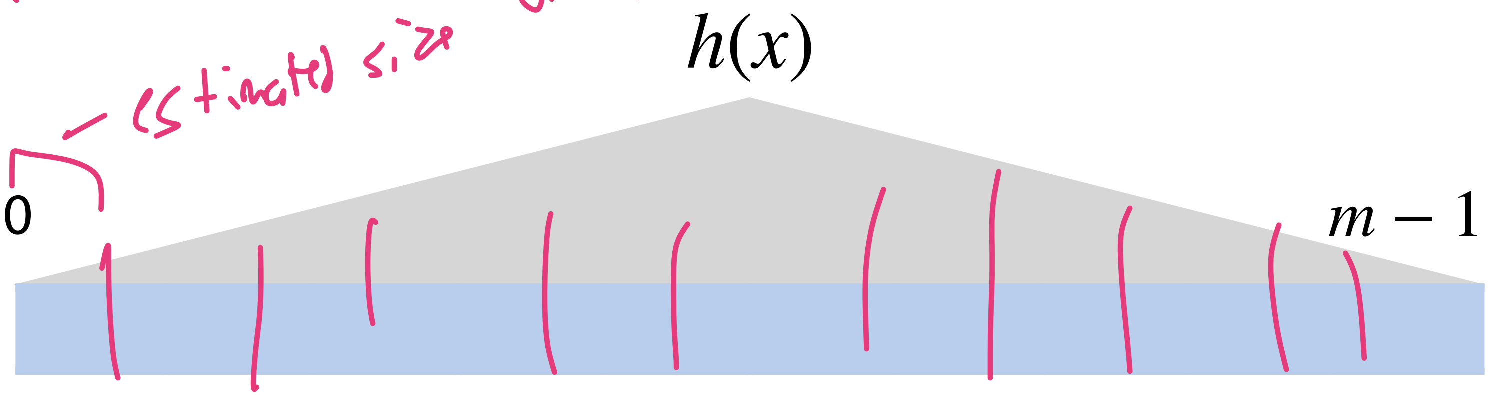
Imagine we have a SUHA hash h over a range m.

Inserting a new key is equivalent to adding a card to our hat!

Tracking only the minimum value is a **sketch** that estimates the cardinality!

Why $\frac{1}{N+1}$ (later slide)?

estimated size of partition



N items across range, N+1 partitions

Cardinality Estimation

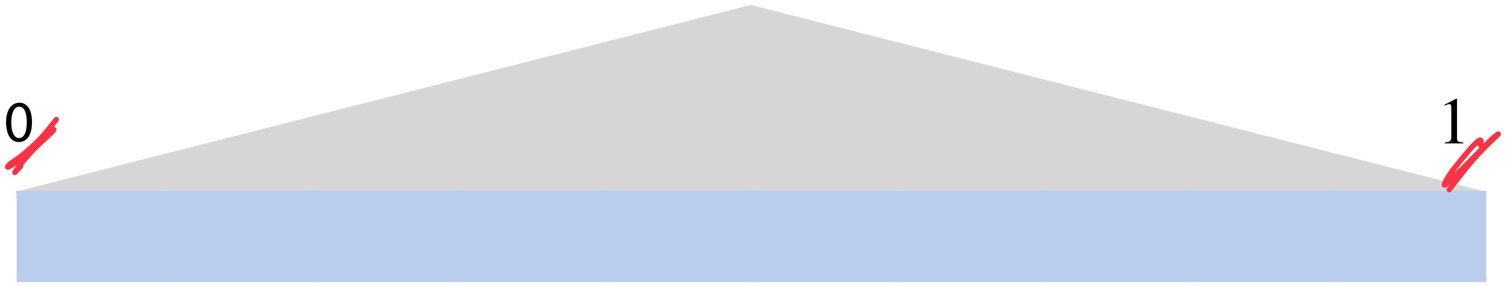
Imagine we have a SUHA hash h over a range m.

Inserting a new key is equivalent to adding a card to our hat!

Tracking only the minimum value is a **sketch** that estimates the cardinality!

To make the math work out, lets normalize our hash…

$$h'(x) = h(x) \;/\; (m - 1)$$

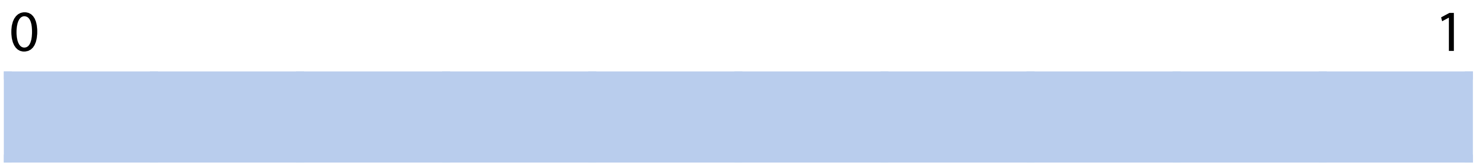


Probability of being minima

Cardinality Sketch

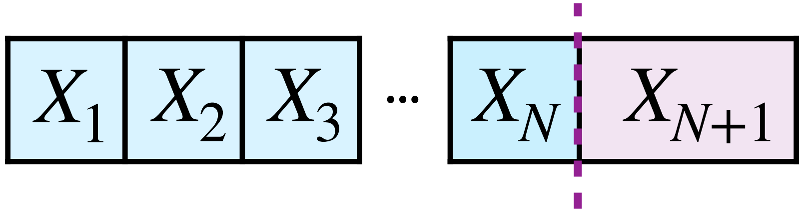
Let $M = min(X_1, X_2, \ldots, X_N)$ where each $X_i \in [0, 1]$ is an uniform independent random variable

Claim: $\mathbf{E}[M] = \dfrac{1}{N+1}$



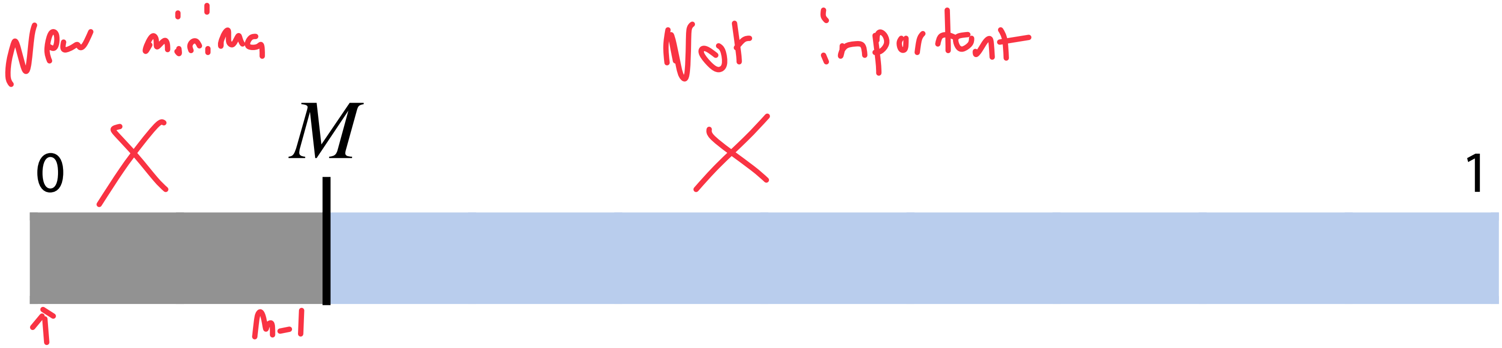
Cardinality Sketch

Consider an $N+1$ draw:



$$M = \min_{1 \le i \le N} X_i$$

X_{N+1} can end up in one of two ranges:



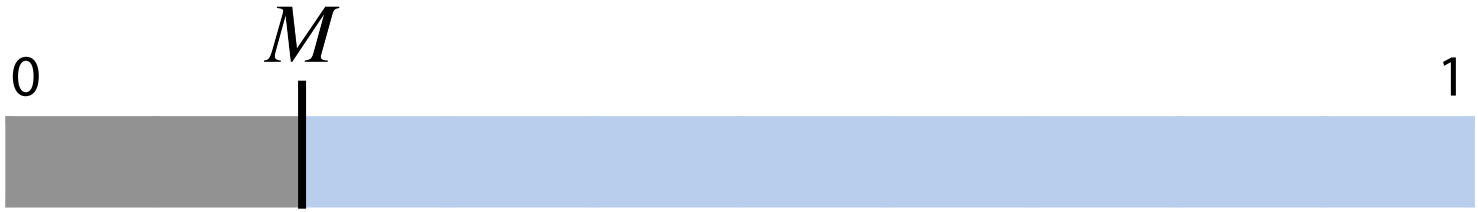
Cardinality Sketch

Consider an $N+1$ draw: $X_1 \; X_2 \; X_3 \; \cdots \; X_N \; X_{N+1}$

$$M = \min_{1 \le i \le N} X_i$$

X_{N+1} can end up in one of two ranges:

X_{N+1} will be the new minimum with probability M



Cardinality Sketch

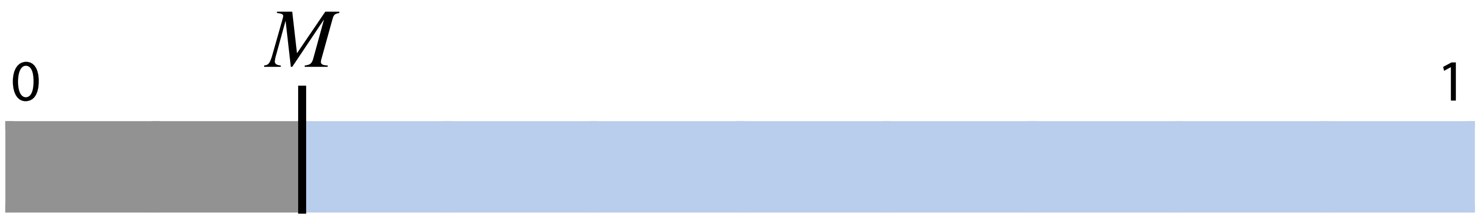
Consider an $N + 1$ draw: X_1 | X_2 | X_3 ... X_N | X_{N+1}

$$M = \min_{1 \leq i \leq N} X_i$$

X_{N+1} can end up in one of two ranges:

X_{N+1} will be the new minimum with probability M

X_{N+1} will not change minimum with probability $1 - M$



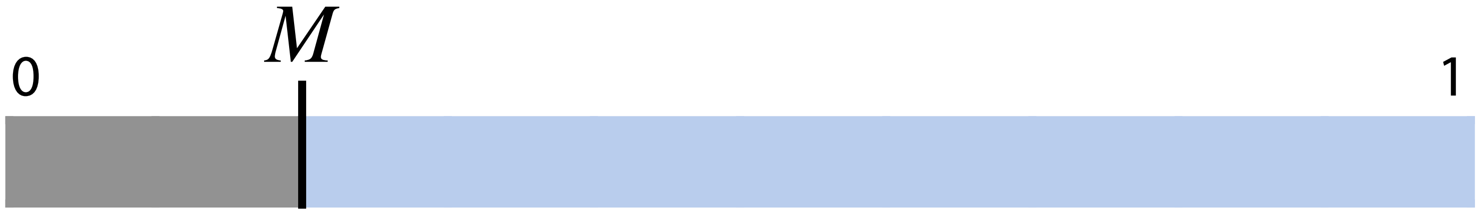
Cardinality Sketch

Consider an $N+1$ draw: $X_1 \; X_2 \; X_3 \; \cdots \; X_N \; X_{N+1}$

$$M = \min_{1 \le i \le N} X_i$$

X_{N+1} **will be the new minimum with probability** M

By definition of SUHA, X_{N+1} has a $\dfrac{1}{N+1}$ chance of being smallest item



Cardinality Sketch

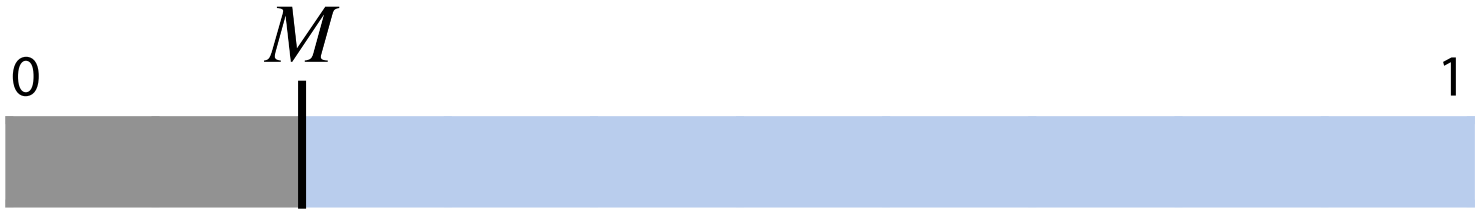
Consider an $N+1$ draw: $\boxed{X_1}\boxed{X_2}\boxed{X_3} \cdots \boxed{X_N}\boxed{X_{N+1}}$

$$M = \min_{1 \le i \le N} X_i$$

X_{N+1} **will be the new minimum with probability** M

By definition of SUHA, X_{N+1} has a $\dfrac{1}{N+1}$ chance of being smallest item

Thus, $\mathbf{E}[M] = \dfrac{1}{N+1}$



Cardinality Sketch

Still $O(N)$ b/c process / hash each item
but space is $O(1)$

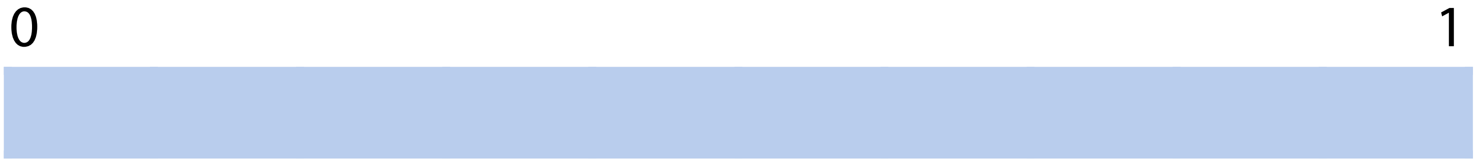
Claim: $\mathbf{E}[M] = \dfrac{1}{N+1}$ $\qquad N \approx \dfrac{1}{M} - 1$

Actual N is 5

Attempt 1	0.962	0.328	0.771	0.952	0.923	$N = 2.05$
Attempt 2	0.253	0.839	0.327	0.655	0.491	$N = 2.953$
Attempt 3	0.134	0.580	0.364	0.743	0.931	$N = 6.5$

Cardinality Sketch

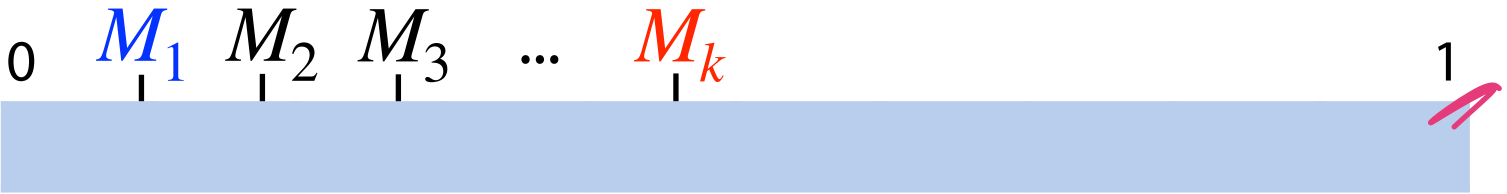
The minimum hash is a valid sketch of a dataset but can we do better?



Cardinality Sketch

Claim: Taking the k^{th}-smallest hash value is a better sketch!

Claim: $\mathbf{E}[M_k] = \dfrac{k}{N+1}$



Cardinality Sketch

Claim: Taking the k^{th}-smallest hash value is a better sketch!

Claim: $\dfrac{\mathbf{E}[M_k]}{k} = \dfrac{1}{N+1}$

$$= \left[\mathbf{E}[M_1] + (\mathbf{E}[M_2] - \mathbf{E}[M_1]) + \ldots + (\mathbf{E}[M_k] - \mathbf{E}[M_{k-1}])\right] \cdot \frac{1}{k}$$

$M_3 - M_2$

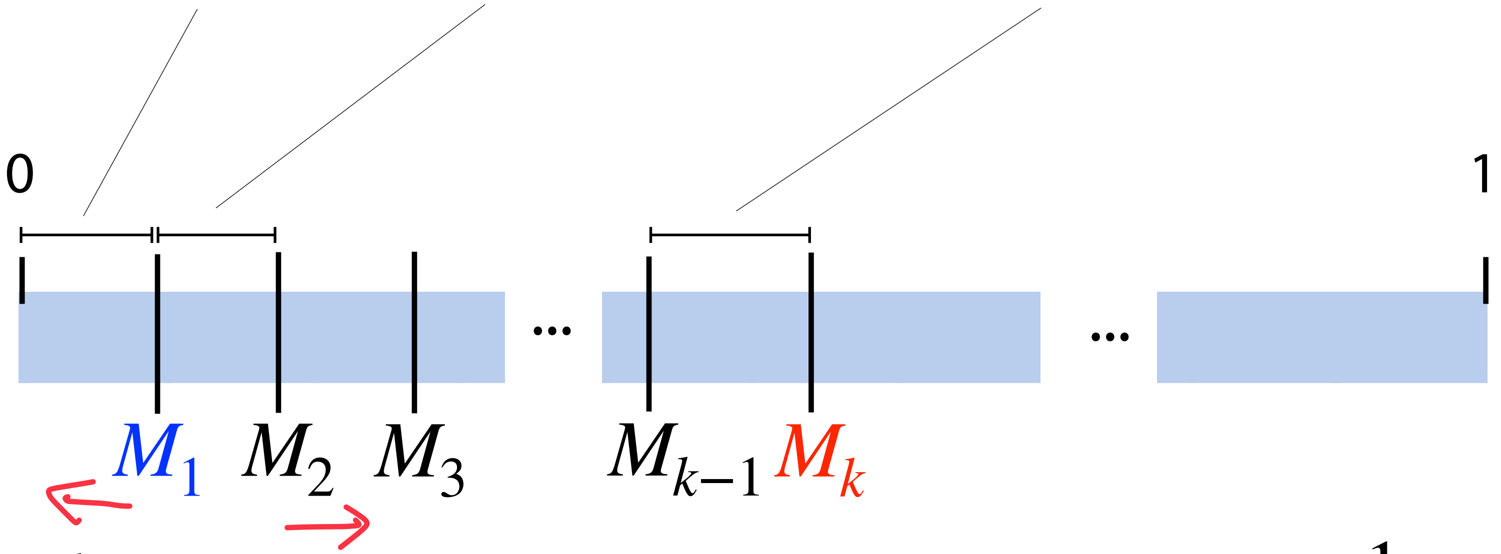
$M_2 - M_1$

M_1 M_2 M_3 ... M_{k-1} M_k

Cardinality Sketch

$$\frac{1}{N+1} = \frac{\mathbf{E}[M_k]}{k}$$

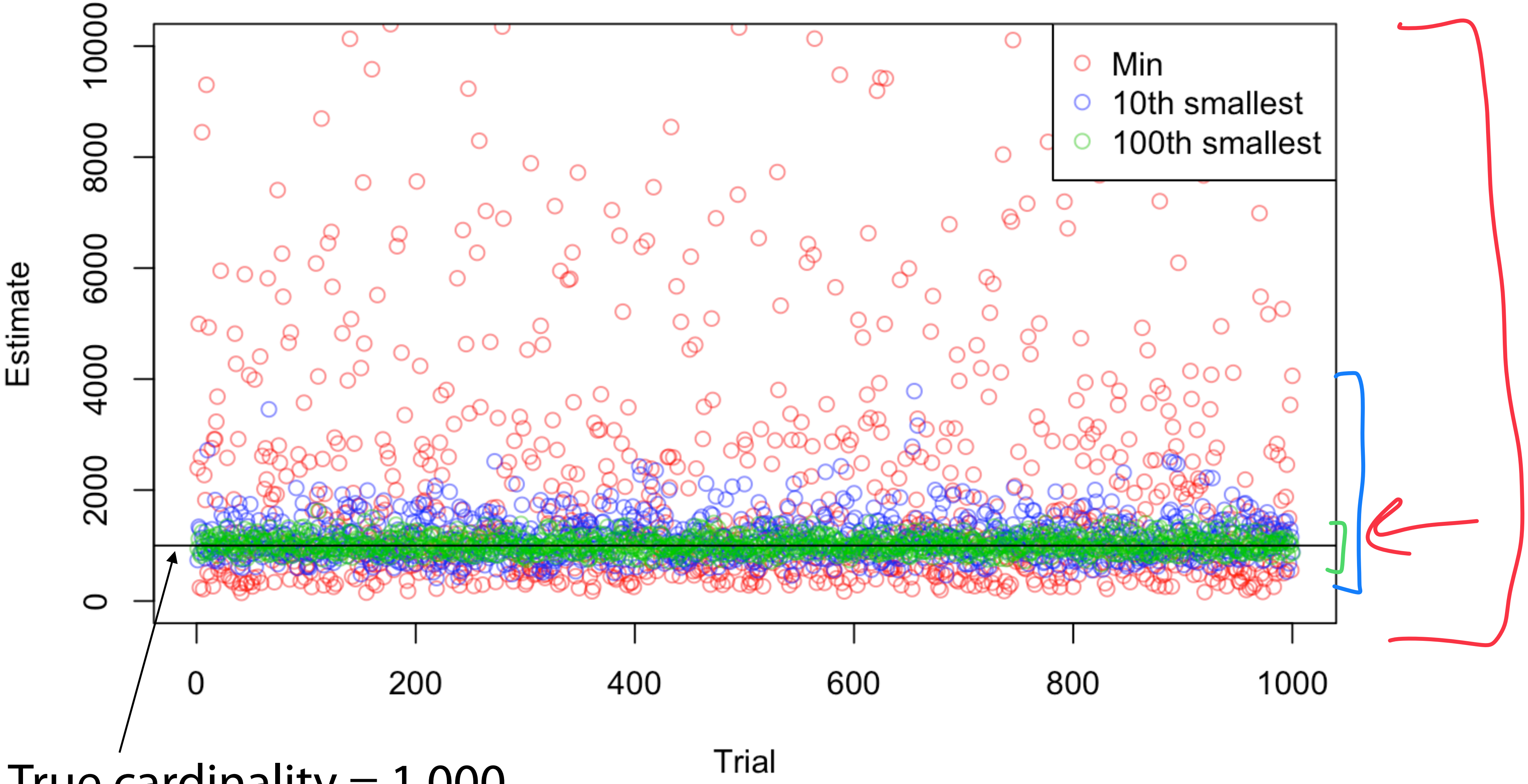
$$= \Big[\underbrace{\mathbf{E}[M_1]} + \underbrace{(\mathbf{E}[M_2] - \mathbf{E}[M_1])} + \ldots + \underbrace{(\mathbf{E}[M_k] - \mathbf{E}[M_{k-1}])}\Big] \cdot \frac{1}{k}$$



k^{th} minimum value (KMV)

Averages k estimates for $\dfrac{1}{N+1}$

Cardinality Sketch



True cardinality = 1,000

Cardinality Sketch

Given any dataset and a SUHA hash function, we can **estimate the number of unique items** by tracking the **k-th minimum hash value**.

Hash

1st min		2nd		3rd
0.253	0.839	0.327	0.655	0.491

$\frac{1}{.253} - 1 = 3$

$\frac{2}{.327} - 1 = 5.1$

$\frac{3}{.491} - 1 = 5.1$

To use the k-th min, we have to track k minima. **Can we use ALL minima?**

Tradeoff, larger K better accuracy but size cost goes up $K = n/2$ is $O(n)$

Applied Cardinalities

Cardinalities

$$|A|$$

$$|B|$$

$$|A \cup B|$$

$$|A \cap B|$$

Set similarities

$$O = \frac{|A \cap B|}{min(|A|, |B|)}$$

$$J = \frac{|A \cap B|}{|A \cup B|}$$

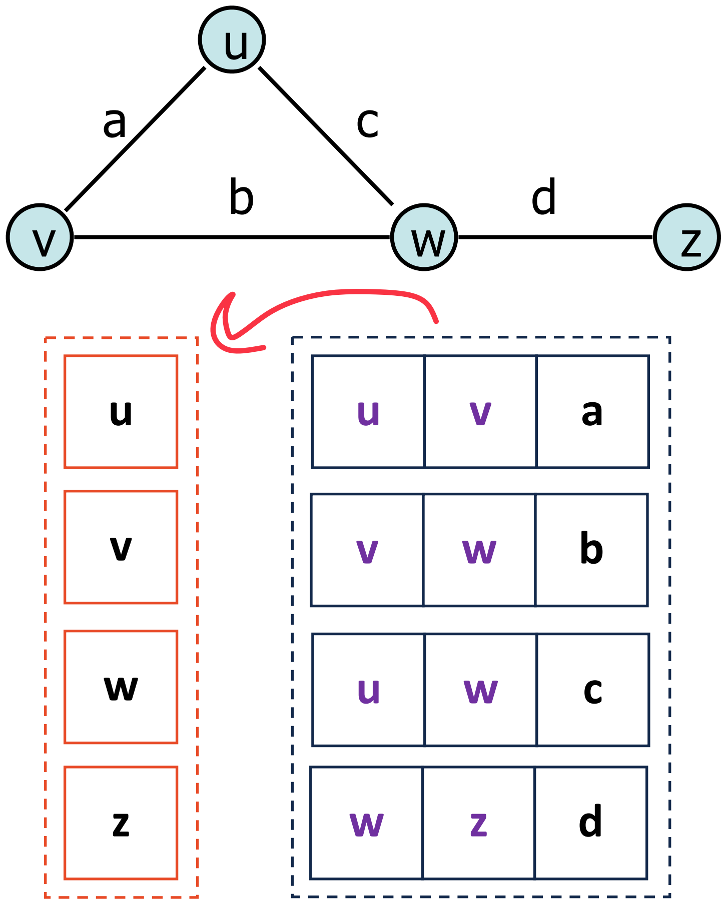
Real-world Meaning

```
AGGCCACAGTGTATTATGACTG
|||||||||||  |||||||||
AGGCCACAGTGAGTTATGACTG

AAAAAAAAAAAGATGT-AAGTA
|||||||||||||||| |||||
AAAAAAAAAAAGATGTAAAGTA

GAGG--TCAGATTCACAGCCAC
||||  ||||||||||||||||
GAGGGGTCAGATTCACAGCCAC
```

Review content (if time)

Graph Implementation: Edge List $|V| = n, |E| = m$

The equivalent of an 'unordered' data structure



Vertex Storage:

An optional list of vertices

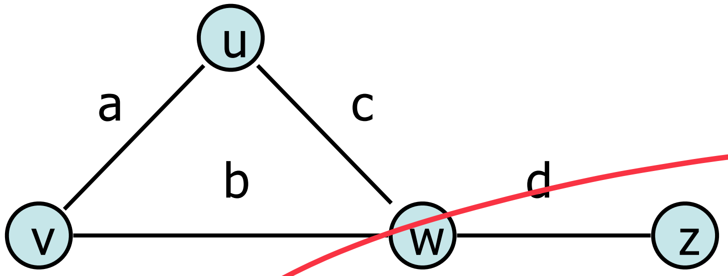
Edge Storage:

A list storing edges as (V1, V2, Weight)

Most graphs are stored as just an edge list!

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$



Vertex Storage:

A hash table of vertices

Implicitly or explicitly store index

or,

u	0
v	1
w	2
z	3

u v w z

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

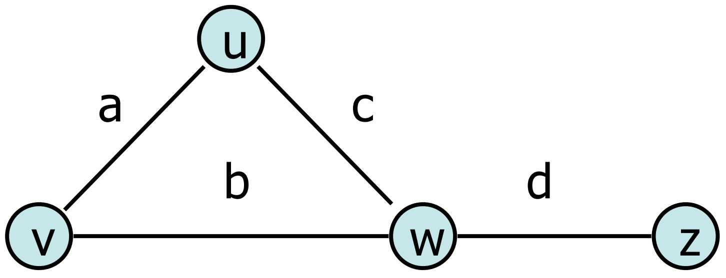
Edge Storage:

A |V| x |V| matrix of edges

Weight is stored at position (u, v)

:(we store things we don't have

Adjacency List



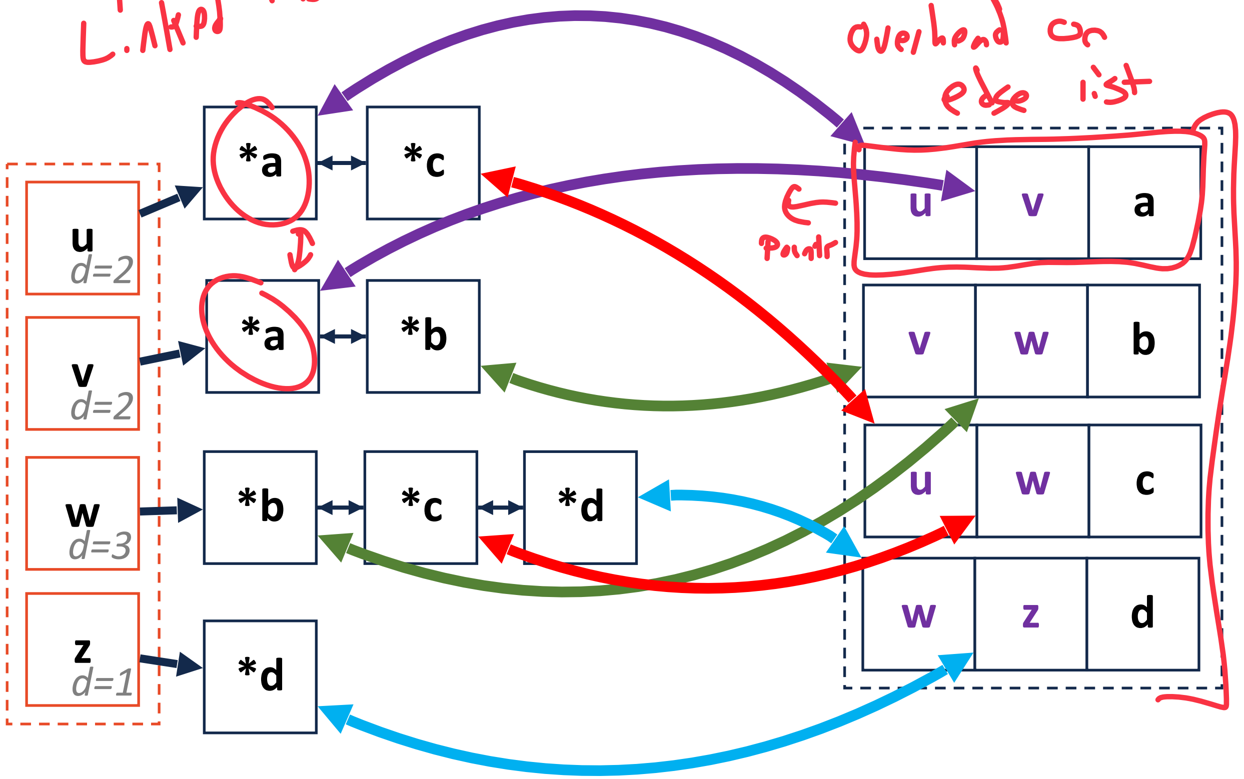
Vertex Storage:

A bidirectional linked list with size variable

Each node is a pointer to edge in edge list

Linked list

Overhead on edge list

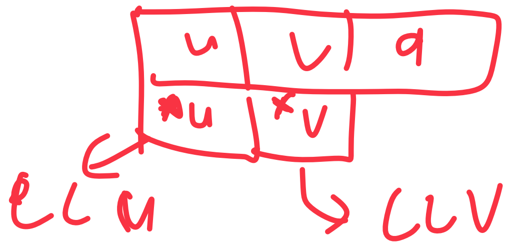


Edge Storage:

A list of (v1, v2, weight) edges

Also store pointers back to nodes

Make access fast!



$|V| = n, |E| = m$

Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	n+m	n	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
incidentEdges(v)	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

$1 \le deg(v) \le n$

Summary: DFS and BFS

$|V| = n, |E| = m$

Both are **O(n+m)** traversals! They label every edge and every node

BFS

Solves unweighted MST

Solves shortest path *

Solves cycle detection

Memory bounded by width

DFS

Solves unweighted MST

Solves cycle detection

Memory bounded by longest path

↳ Make sure you understand start position!

Kruskal's Algorithm

```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T
```

1) Build a **priority queue** on edges

A minheap or *A sorted array*

2) Build a **disjoint set** on vertices

All vertices start as their own set

3) Loop through min edges

If edge connects two disjoint sets

Union sets and record edge in MST

4) Stop when:

N-1 edges recorded

Only a single disjoint set remains

Kruskal's Algorithm

```
1   KruskalMST(G):
2     DisjointSets forest
3     foreach (Vertex v : G.vertices()):
4       forest.makeSet(v)
5
6     PriorityQueue Q    // min edge weight
7     Q.buildFromGraph(G.edges())
8
9     Graph T = (V, {})
10
11    while |T.edges()| < n-1:
12      Vertex (u, v) = Q.removeMin()
13      if forest.find(u) != forest.find(v):
14         T.addEdge(u, v)
15         forest.union( forest.find(u),
16                       forest.find(v) )
17
18    return T
19
```

$|V| = n, |E| = m$

What is the Big O?

2 — 4: O(n)

6 — 7: Heap: O(m)
Sorted List: O(m log m)

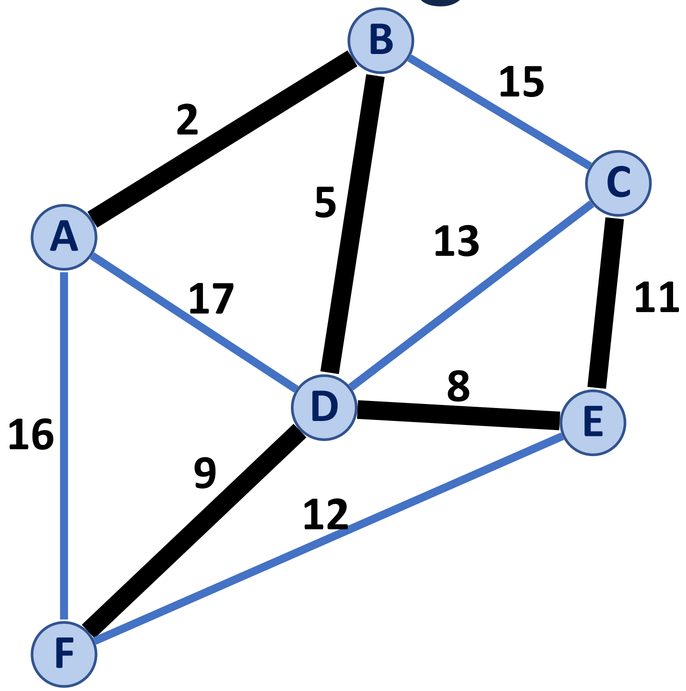
11: m x <12-17>

12—17: Heap: O(log m)
Sorted List: O(1)

O(n + m + m log m)

Simplified: O(n + m log n)

Prim’s Algorithm



A	B	C	D	E	F
0, —	2, A	11, E	5, B	8, D	9, D

```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

  return T
```

Prim's Big O

|V|= n, |E|= m

7 — 9: O(n)

12—14:

MinHeap: O(n)

Unsorted Array: O(1)

16—22: Complicated!

```
6   PrimMST(G, s):
7     foreach (Vertex v : G.vertices()):
8       d[v] = +inf
9       p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T           // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

Depends on choice of **PriorityQueue** (MinHeap vs Unsorted Array)

Depends on choice of **Graph** (Adjacency Matrix vs Adjacency List)

Prim's Algorithm

Sparse Graph: (m ~ n)

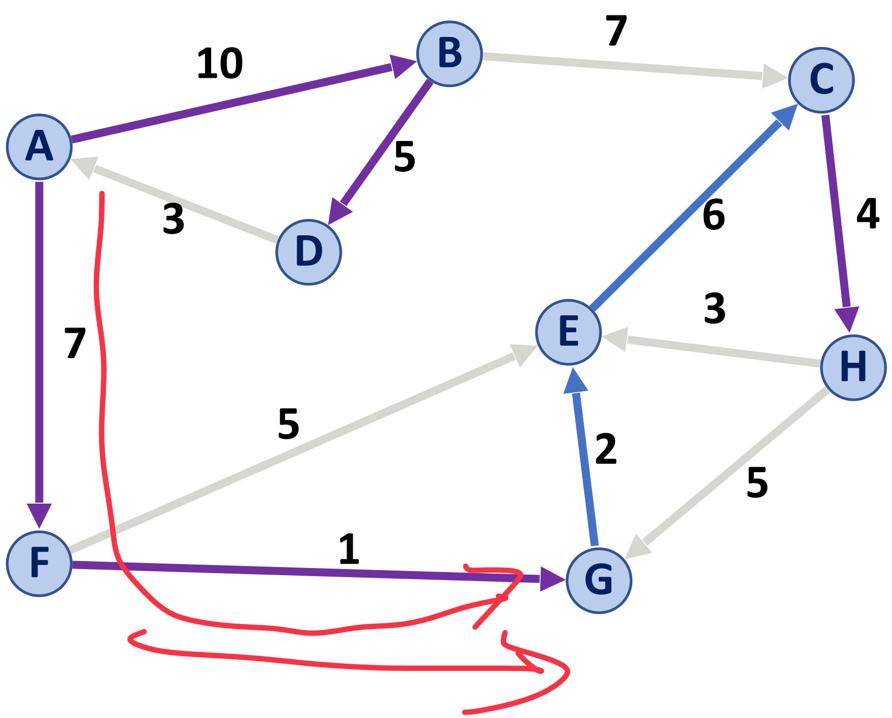
Dense Graph: (m ~ n^2)

```
6   PrimMST(G, s):
7     foreach (Vertex v : G.vertices()):
8       d[v] = +inf
9       p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T          // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

Lines 7 — 14 are O(n) [at most]

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m \lg(n))$	$O(n \lg(n) + m \lg(n))$
Unsorted Array	$O(n^2)$	$O(n^2)$

Dijkstra's Algorithm (SSSP)




```
DijkstraSSSP(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = u
```

* only difference

A	B	C	D	E	F	G	H
--	A	E	B	G	A	F	C
0	10	16	15	10	7	8	20

Floyd-Warshall Algorithm

Floyd-Warshall's Algorithm is an alternative to Dijkstra in the presence of negative-weight edges (not negative weight cycles).

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex u : G):
    foreach (Vertex v : G):
      foreach (Vertex w : G):
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]
```

A Hash Table based Dictionary

User Code (is a map):

```
Dictionary<KeyType, ValueType> d;
d[k] = v;
```

A **Hash Table** consists of three things:

1. A hash function

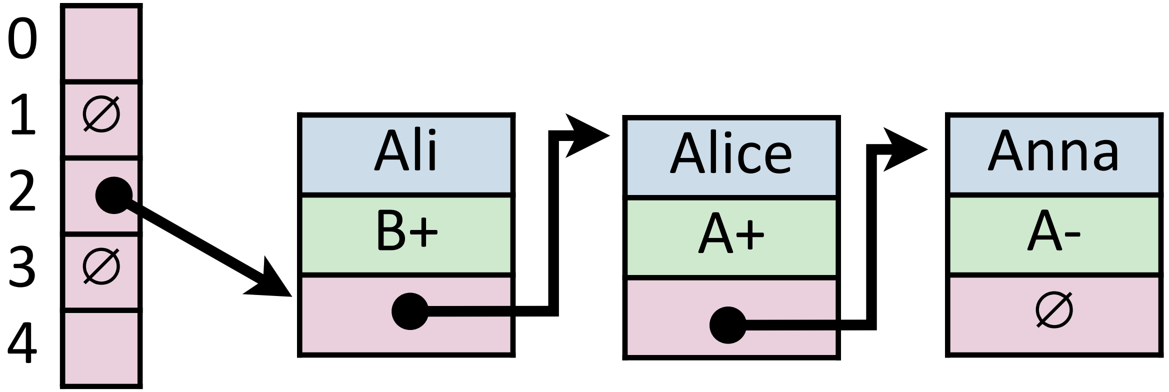
2. A data storage structure

3. A method of addressing *hash collisions*

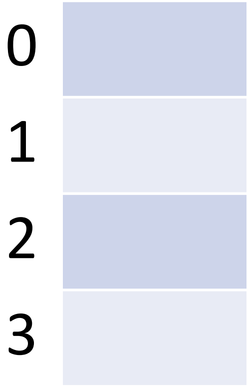
Open vs Closed Hashing

Addressing hash collisions depends on your storage structure.

- **Open Hashing:** store *k,v* pairs externally



- **Closed Hashing:** store *k,v* pairs in the hash table



Linear prob

double prob

Separate Chaining Under SUHA

Claim: Under SUHA, expected length of chain is $\dfrac{n}{m}$

Table Size: m

Num objects: n

$\alpha_j =$ expected # of items hashing to position j

$$\alpha_j = \sum_i H_{i,j}$$

$$H_{i,j} = \begin{cases} \text{1 if item i hashes to j} \\ \text{0 otherwise} \end{cases}$$

$$E[\alpha_j] = E[\sum_i H_{i,j}]$$

$$Pr[H_{i,j} = 1] = \frac{1}{m}$$

$$E[\alpha_j] = n * Pr(H_{i,j} = 1)$$

$$\boxed{\mathbf{E}[\alpha_\mathbf{j}] = \frac{\mathbf{n}}{\mathbf{m}}}$$

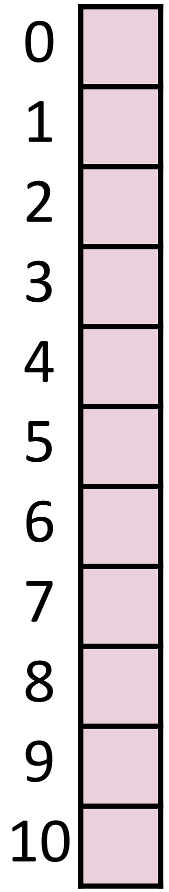
Separate Chaining Under SUHA

Under SUHA, a hash table of size *m* and *n* elements:

Find runs in: $O(1+\alpha)$

Insert runs in: $O(1)$

Remove runs in: $O(1+\alpha)$



Running Times *(Don't memorize these equations, no need.)*

The expected number of probes for find(key) under SUHA

Linear Probing:

- Successful: **$\frac{1}{2}(1 + 1/(1-\alpha))$**
- Unsuccessful: **$\frac{1}{2}(1 + 1/(1-\alpha))^2$**

Double Hashing:

- Successful: **$1/\alpha * \ln(1/(1-\alpha))$**
- Unsuccessful: **$1/(1-\alpha)$**

Separate Chaining:

- Successful: **$1 + \alpha/2$**
- Unsuccessful: **$1 + \alpha$**

Instead, observe:

- As α increases:

Runtime approaches infinity!

- If α is constant:

Runtime is a constant!

Resizing a hash table

When and how do you resize?

↳ rehashing everything

*

Before capacity

.7 – 0.9

Any (review) questions?