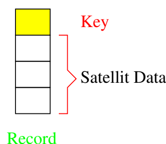
Hashing

AA-GEI FIB, UPC

Spring 2025-2026

Data Structures: Reminder

Given a universe $\mathcal{U}$, a dynamic set of records, where each record has a unique key:



- Array
- Linked List (and variations)
- Stack (LIFO): Supports push and pop
- Queue (FIFO): Supports enqueue and dequeue
- Deque: Supports push, pop, enqueue and dequeue
- Priority Queue: Supports insertions, deletions, find Max/MIN
- Hash tables

Data structures for dynamic sets

DICTIONARY

Data structure for maintaining $\mathcal{S} \subset \mathcal{U}$ together with operations:

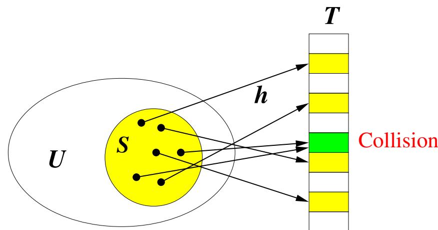
- Search(k): decide if $k \in \mathcal{S}$
- Insert(k): $\mathcal{S} := \mathcal{S} \cup \{k\}$
- Delete(k): $\mathcal{S} := \mathcal{S} \setminus \{k\}$

Hashing

Data Structure that supports *dictionary* operations on an universe of numerical keys.

- There are $2^{63} = 18446744073709551616$. possible keys represented as 64-bit integers.
- Make a tradeoff *time/space*
- Define a hash table $T[0, \ldots, m-1]$ by a hash function $h : \mathcal{U} \rightarrow T[0, \ldots, m-1]$

Hans P. Luhn (1896-1964)

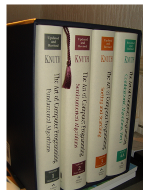


Simple uniform hashing function.

- We want to store a maximum of n keys in a hashing table T with m slots.
- The performance of hashing depends on how well h distributes the keys on the m slots.
- h is simple uniform if it hash any key *with equal probability* into any slot, independently of where other keys go.
- In this way, we get a load factor $\alpha = n/m$, the average number of keys per slot.

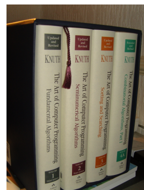
How to choose h?

For an exhaustive treaty on Hashing: D. Knuth, Vol. 3 of *The Art of computing programming*



How to choose h?

For an exhaustive treaty on Hashing: D. Knuth, Vol. 3 of *The Art of computing programming*



h depends on the type of key:

- For keys in the real interval $[0, 1)$, we can use $h(k) = \lfloor mk \rfloor$.
- For keys in the real interval $[s, t)$ scale by $1/(t - s)$, and use the previous method, $h(k/(t - s)) = \lfloor mk/(t - s) \rfloor$.

The division method

Choose m prime or as far as possible from a power of 2,

$$h(k) = k \mod m$$

Fast ($\Theta(1)$) to compute, in most languages ($k\%m$)

Be aware: if $m = 2^r$ the hash function does not depend on all the bits of k

If $r = 6$ with $k = 1011000111\underbrace{011010}_{=h(k)}$

$(45530 \mod 64 = 858 \mod 64)$

How to deal with large n?

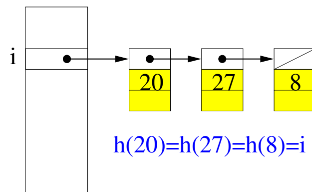
For large n, to compute $h = n \mod m$, we can use mod arithmetic + Horner's method:

$$\begin{aligned}&((((((((((97 \cdot 128 + 118) \cdot 128 + 101) \cdot 128 + 114) \cdot 128 + 121)\\&\cdot 128 + 111) \cdot 128 + 110) \cdot 128 + 103) \cdot 128 + 107)\\&\cdot 128 + 101) \cdot 128 + 121 \mod m\\&= (((((((((\underbrace{\underbrace{(97 \cdot 128 + 118 \mod m)}_{} \cdot 128) \mod m + 101)}_{} \cdot \ldots)))))))\end{aligned}$$

Collision resolution: Separate chaining

For each table address, construct a linked list of the items whose keys hash to that address.

- Every key goes to the same slot
- Time to explore the list = length of the list



Cost of average analysis of chaining

The cost of the dictionary operations using hashing:

- Insertion of a new key: $\Theta(1)$.
- Search of a key: O(length of the list)
- Deletion of a key: O(length of the list).

Under the hypothesis that h is *simply uniform*, each key x is equally likely to be hashed to any slot of T, independently of where other keys are hashed

Therefore, the expected number of keys falling into $T[i]$ is $\alpha = n/m$.

Cost of search

- For an unsuccessful search (x is not in T), we have to explore the list at $h(x) \to T[i]$. So, the expected time to search the list at $T[i]$ is $O(1+\alpha)$.
  (α of searching the list and $\Theta(1)$ of computing $h(x)$ and going to slot $T[i]$)

Cost of search

- For an unsuccessful search (x is not in T), we have to explore the list at $h(x) \to T[i]$. So, the expected time to search the list at $T[i]$ is $O(1+\alpha)$.
  (α of searching the list and $\Theta(1)$ of computing $h(x)$ and going to slot $T[i]$)
- For an successful search, we obtain the same bound, although in most of the cases we would have to search a fraction of the list until finding the x element.)

Cost of search

- For an unsuccessful search (x is not in T), we have to explore the list at $h(x) \to T[i]$. So, the expected time to search the list at $T[i]$ is $O(1+\alpha)$.
  (α of searching the list and $\Theta(1)$ of computing $h(x)$ and going to slot $T[i]$)
- For an successful search, we obtain the same bound, although in most of the cases we would have to search a fraction of the list until finding the x element.)
- Under the assumption of simple uniform hashing, in a hash table with chaining, a search takes time $\Theta(1+\frac{n}{m})$ on average.

Cost of search

- For an unsuccessful search (x is not in T), we have to explore the list at $h(x) \to T[i]$. So, the expected time to search the list at $T[i]$ is $O(1+\alpha)$.
  (α of searching the list and $\Theta(1)$ of computing $h(x)$ and going to slot $T[i]$)
- For an successful search, we obtain the same bound, although in most of the cases we would have to search a fraction of the list until finding the x element.)
- Under the assumption of simple uniform hashing, in a hash table with chaining, a search takes time $\Theta(1+\frac{n}{m})$ on average.
- Notice that if $n = \Theta(m)$ then $\alpha = O(1)$ and search time is $\Theta(1)$.

Universal hashing: Motivation

- For every deterministic hash function, there is a set of bad instances.
- An adversary can arrange the keys so your function hashes most of them to the same slot.

Universal hashing: Motivation

- For every deterministic hash function, there is a set of bad instances.
- An adversary can arrange the keys so your function hashes most of them to the same slot.
- Create a set $\mathcal{H}$ of hash functions on $\mathcal{U}$ and choose a hash function at random and independently of the keys.
- The adversary might known the probability space but not the particular selection.

Universal hashing

Let $\mathcal{U}$ be the universe of keys and let $\mathcal{H}$ be a collection of hash functions for a hash table $T[0, \ldots, m-1]$, $\mathcal{H}$ is universal if $\forall x, y \in \mathcal{U}, x \neq y$, then

$$|\{h \in \mathcal{H} \mid h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{m}.$$

In an equivalent way, $\mathcal{H}$ is *universal* if $\forall x, y \in \mathcal{U}, x \neq y$, and for any h chosen uniformly from $\mathcal{H}$, we have

$$\mathbf{Pr}\left[h(x) = h(y)\right] \leq \frac{1}{m}.$$

Universality gives good average-case behaviour

Theorem

Pick u.a.r. h from a universal hash family $\mathcal{H}$ and build a table with size m for a set of n keys. For any key x, let C_x be a random variable counting the number of collisions with others keys y in T. Then,

$$\mathbf{E}\left[C_x\right] \leq n/m.$$

Universality gives good average-case behaviour

Proof.

Note that the expectation is over the choice of $h \in \mathcal{H}$ not over any distribution of .

We want to compute the expected size of list at $T[i]$.

For each pair of different keys x and y, define the collision indicator r.v.

$$Z_{xy} = \begin{cases} 1 & \text{if } h(x) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

Then, by definition $\mathbf{E}[Z_{xy}] \leq 1/m$ and $Z_x = \sum_{y \in T - \{x\}} Z_{xy}$

Universality gives good average-case behaviour

For each key x define the r.v. Y_x that counts the number of other keys that collide with x in position $h(x)$.

$$\mathbf{E}[Y_x] = \mathbf{E}\left[\sum_{y \neq x} Z_{xy}\right] = \sum_{y \neq x} \mathbf{E}[Z_{xy}] \leq \sum_{y \neq x} 1/m = \frac{n-1}{m}$$

EndProof.

Therefore, universal hash functions dismount adversarial strategy

Construction of a universal family: $\mathcal{H}$

Let $\mathcal{U}$ be the key universe and let N be the maximum key value. Our target is a hash table with m positions, $T[0,\ldots,m-1]$.

- *Choose a prime p, $N \leq p \leq 2N$. Then $\mathcal{U} \subset \mathbb{Z}_p = \{0,1,\ldots,p-1\}$.*
- Define $\mathcal{H} = \{h_{a,b} | a,b \in \mathbb{Z}_p, a \neq 0\}$.

Construction of a universal family: $\mathcal{H}$

Let $\mathcal{U}$ be the key universe and let N be the maximum key value. Our target is a hash table with m positions, $T[0, \dots, m-1]$.

- *Choose a prime p,* $N \leq p \leq 2N$*. Then* $\mathcal{U} \subset \mathbb{Z}_p = \{0, 1, \dots, p-1\}$*.*
- Define $\mathcal{H} = \{h_{a,b} | a, b \in \mathbb{Z}_p, a \neq 0\}$.
- To select u.a.r. $h \in \mathcal{H}$, *choose independently and u.a.r.* $a \in \mathbb{Z}_p^+$ *and* $b \in \mathbb{Z}_p$*. Given a key x define* $h_{a,b}(x) = (\underbrace{(ax+b) \mod p}_{g_{a,b}(x)}) \mod m$*.*

Construction of a universal family: $\mathcal{H}$

Let $\mathcal{U}$ be the key universe and let N be the maximum key value. Our target is a hash table with m positions, $T[0,\dots,m-1]$.

- *Choose a prime p, $N \leq p \leq 2N$. Then $\mathcal{U} \subset \mathbb{Z}_p = \{0,1,\dots,p-1\}$.*
- Define $\mathcal{H} = \{h_{a,b} | a,b \in \mathbb{Z}_p, a \neq 0\}$.
- To select u.a.r. $h \in \mathcal{H}$, *choose independently and u.a.r. $a \in \mathbb{Z}_p^+$ and $b \in \mathbb{Z}_p$. Given a key x define $h_{a,b}(x) = \underbrace{((ax+b) \mod p)}_{g_{a,b}(x)} \mod m$.*
- Example: $p = 17, m = 6$, we have $\mathcal{H}_{17,6} = \{h_{a,b} : a \in \mathbb{Z}_p^+, b \in \mathbb{Z}_p\}$
  if $x = 8$, $a = 3, b = 4$ then
  $h_{3,4}(8) = ((3 \cdot 8 + 4) \mod 17) \mod 6 = 5$

Properties of $\mathcal{H}$

1. $h_{ab} : \mathbb{Z}_p \to \mathbb{Z}_m$.
2. $|\mathcal{H}| = p(p-1)$. (We can select a in $p-1$ ways and b in p ways)
3. Specifying an $h \in \mathcal{H}$ requires $O(\lg p) = O(\lg N)$ bits.
4. To choose $h \in \mathcal{H}$ select a, b independently and u.a.r. from $\mathbb{Z}_p^+$ and $\mathbb{Z}_p$.
5. Evaluating $h(x)$ is fast.

Theorem

The family $\mathcal{H}$ is universal.

Proof.

- Consider two distinct keys $x, y \in \mathbb{Z}_p$.
- For a given hash function h_{ab} let $r = (ax + b) \mod p$ and $s = (ay + b) \mod p$.
- $r - s \equiv a(x - y) \mod p$. As p is prime, and both left terms non zero, $r \neq s$.
- Each of the possible $p(p-1)$ choices of the pair (a, b) with $a \neq 0$ yields a different resulting pair (r, s).

- Then, there is a one-to-one correspondence among (a, b) and (r, s).
- So, the probability that two distinct keys collide is the probability that $r \equiv s \mod m$, when r and s are randomly chosen as distinct values modulo p.
- For a given value of r, of the $p - 1$ possible values for s, the number of values s with $s \neq r$ and $s \equiv r \mod p$ is at most $\lceil p/m \rceil - 1 \leq (p - 1)/m$.
- Therefore, $\Pr h_{ab}(x) = h_{ab}(y) \leq 1/m$.

EndProof.

Bloom filter

Given a set of elements S, we want a data structure for supporting insertions and querying about membership in S.

In particular, we wish a DS s.t.

- *minimizes* the use of memory,
- *can check membership* as fast as possible.

Burton Bloom: The Bloom filter data structure. Comm. ACM, July 1970.
A hash data structure where each register in the table is one bit

Query on a list of e-mails

- We have a set S of 10^9 e-mail addresses, where the typical e-mail address is 20 bytes.
- It does not seem reasonable to store S in main memory. We can spare 1 Gigabyte of memory, which is approximately 10^9 bytes or 8×10^9 bites.
- How to keep information about S in main memory and be able to query it?

Definition Bloom filter

- Giving a set $S = \{x_1, \dots, x_n\}$.
- Create a one bit hash table $T[0, \dots, m-1]$, and take a hash function $h : S \to [0 \dots m-1]$ from a universal family.
- Initially all m bits stored in T are set to 0.

Create table Insert (x)

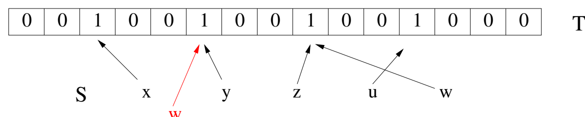
$T[h(x)] = 1$

Notice: once we have hashed S into T, we erase S.

Query table inS(y)

if $T[h(y)] == 1$ **then**
  return Yes
else
  return No
end if

False positives



- Bloom filter needs $O(m)$ space.
- Answers membership queries in $\Theta(1)$.
- Easy to augment but no support for removals.
- In a query $y \in S$?, a Bloom filter always
  - reports correctly if $y \in S$.
  - but might not if $y \notin S$. It may be the case that $T[h(y)] = 1$, we can have false positives.
- How large is the error of having a false positive?

Probability of having a false positives

- Let $|S| = n$, we constructed a BF $(h, T[m])$ hashing the elements in S.
- If we query about $y \in S$?, with $y \notin S$, what is the probability that $T[h(y)] = 1$?
- After all the elements of S are hashed into the Bloom filter, the probability that a specific $T[i] = 0$ is

$$\left(1 - \frac{1}{m}\right)^n \sim e^{-n/m}$$

- Therefore, for a $y \notin S$, the probability of false positive p:

$$p = \mathbf{Pr}\left[T[h(y)] = 1\right] = 1 - \left(1 - \frac{1}{m}\right)^n \sim 1 - e^{-n/m}.$$

- To minimise p, we have to maximize $e^{-n/m}$
  $\Rightarrow \frac{n}{m}$ has to be small, i.e, $m > n$.

Extension: Amplify

- Take k different uniform hash functions $\{h_0, h_2, \ldots, h_{k-1}\}$ in the same 2-universal hash family.
- Create a boolean array $T[k, m]$ initally set to 0
- For each $x \in S$ and $0 \leq j < k$ set $T[j, h_j(x)] = 1$.
- When making a query about if $y \in S$, if one of $T[0, h_0(y)], \ldots T[k-1, h_{k-1}(y)]$ is 0, we know that $y \notin S$.
- otherwise, we accept $y \in S$.

Extension: Amplify

- After hashing S, for an specific pair i, j the probability of having a 0 in position $T[i,j]$ is $(1 - \frac{1}{m})^{kn}$
- The probability f of having a false positive is

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \sim (1 - e^{-kn/m})^{k}.$$

Asymptotic estimations for k

- To minimize the probability of having a false positive: $\frac{\mathrm{d}f}{\mathrm{d}k} = 0$
- Let $f(k) = \ln f$ then $f(k) = k \ln(1 - e^{-kn/m})$
  $\Rightarrow f'(k) = \ln(1 - e^{-kn/m}) + k \frac{ne^{-kn/m}}{m(1-e^{-kn/m})}$
- Making $f'(k) = 0$, we get

$$\boxed{k_{\mathrm{opt}} = \ln 2 \frac{m}{n}}$$

- The probability of having a false positive for k_{opt} is

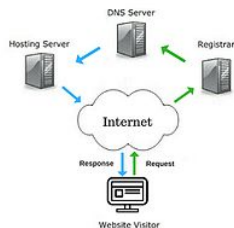
$$f_0 = (1 - e^{\ln 2 \frac{m}{n} \frac{n}{m}})^{\ln 2 \frac{m}{n}} \sim (\frac{1}{2})^{\ln 2 \frac{m}{n}} = 0.6185^{\frac{m}{n}}.$$

Reducing false positive probability

- When n, m are given, to minimize f we need $k_o = (\ln 2)\frac{m}{n}$.
- In this case the false positive probability $f_o = 0.6185^{m/n}$.
- So, when $m = cn$ Bloom filters allow a constant probability of a false positive, i.e. m grows linear wrt n.
- For ex.: if $c = 4$, $f_o = 0.23085$.

An application of Bloom filters: Caching structures

Web server is a computer system that processes requests using http to deliver web pages to clients.

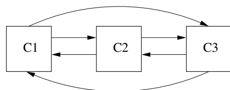


Web cache is a technology for temporary storage of web documents (html pages, images,..) which aim to reduce bandwidth, server load and lag (latency).

Caching structures

Suppose we have a set $\mathcal{U}$ with n URL, each one with 100 characters, i.e in total we have 800n bits.

Consider caches C_1, C_2, C_3, each with documents indexed by their URL.
A query for URL x is sent to one of the caches, that cache must determine which of the caches has x (if x is there)

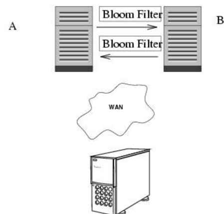


If every C_i stores 10000 documents, that means about 48000000 bits can be exchanged.

Bloom filters may help to reduce the transfer of bits, accepting a small marge of error.

Caching structures

- Each proxy maps all of the URLs in its cache into Bloom Filter.
- Proxies periodically exchange Bloom filters, so queries of other caches can be made locally without sending ICP message.



Cache filtering

Nearly three-quarters of the URLs accessed from a typical web cache are one-hit-wonders accessed by users only once and never again.

To prevent caching one-hit-wonders, a Bloom filter is used to keep track of all URLs that are accessed by users.

A web object is cached only when it has been accessed at least once before.

Markov's inequality

Lemma (Markov's inequality)

If $X \geq 0$ is a r.v, for any constant $a > 0$,

$$\mathbf{Pr}[X \geq a] \leq \frac{\mathbf{E}[X]}{a}.$$

Markov's inequality

Lemma (Markov's inequality)

If $X \geq 0$ is a r.v, for any constant $a > 0$,

$$\mathbf{Pr}\left[X \geq a\right] \leq \frac{\mathbf{E}\left[X\right]}{a}.$$

Corollary

If $X \geq 0$ is a r.v, for any constant $b > 0$,

$$\mathbf{Pr}\left[X \geq b\,\mathbf{E}\left[X\right]\right] \leq \frac{1}{b}.$$

Chebyshev's Inequality

Pafnuty Chebyshev (XIXc)
If you can compute the **Var** [] then you can compute σ and get better bounds for concentration of any r.v. (positive or negative).

Theorem

Let X be a r.v. with expectation μ and standard deviation $\sigma > 0$, then for any $a > 0$

$$\mathbf{Pr}\left[|X - \mu| \geq a\,\sigma\right] \leq \frac{1}{a^2}.$$

Note that $|X - \mu| \geq a\sigma \Leftrightarrow (X \geq a\sigma + \mu) \cup (X \geq \mu - a\sigma)$.

Chernoff Bounds

Sergei Bernstein (1924), Wassily Hoeffding (1964),
Herman Chernoff (1952)

The Chernoff bound can be used when the random variable X is the sum of several independent Poisson trials, where each X_i can has probability of success p_i. The particular case where all p_i are equal is the Bernouilli trials.

Theorem ((Ch-1))

Let $\{X_i\}_{i=0}^n$ be independent Poisson trials, with $\mathbf{Pr}[X_i = 1] = p_i$. Then, if $X = \sum_{i=1}^n X_i$, and $\mu = \mathbf{E}[X]$, we have

1. $\mathbf{Pr}[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu}$, *for* $\delta \in (0,1)$.
2. $\mathbf{Pr}[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$ *for any* $\delta > 0$.

Weak Chernoff's bound, but easy to use

Corollary (Ch-2)

Let $\{X_i\}_{i=0}^{n}$ *be independent Poisson trials, with* $\mathbf{Pr}[X_i = 1] = p_i$*. Then if* $X = \sum_{i=1}^{n} X_i$*, and* $\mu = \mathbf{E}[X]$*, we have*

1. $\mathbf{Pr}[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2}$*, for* $\delta \in (0,1)$*.*
2. $\mathbf{Pr}[X \geq (1+\delta)\mu] \leq e^{-\mu\delta^2/3}$*, for* $\delta \in (0,1]$*.*

An immediate corollary to the previous result:

Corollary (Ch-3)

Let $\{X_i\}_{i=0}^{n}$ *be independent Poisson trials, with* $\mathbf{Pr}[X_i = 1] = p_i$*. Then if* $X = \sum_{i=1}^{n} X_i$*,* $\mu = \mathbf{E}[X]$ *and* $\delta \in (0,1)$*, we have*

$$\mathbf{Pr}[|X-\mu| \geq \delta\mu] \leq 2e^{-\mu\delta^2/3}.$$

Counting the number of distinct elements

Counting the number of distinct elements

- Distinct elements problem: Given a stream s, output $|\{j \mid f_j > 0\}|$. where f_j is the frequency of the j in the stream s

Counting the number of distinct elements

- Distinct elements problem: Given a stream s, output $|\{j \mid f_j > 0\}|$. where f_j is the frequency of the j in the stream s
- In order to solve the problem using sublinear space, we need to use probabilistic algorithms/data structure and some adequate notion of approximation.

An (ϵ, δ)-approximation

An (ϵ, δ)-approximation

- Let $\mathcal{A}(s)$ denote the output of a randomized streaming algorithm $\mathcal{A}$ on input s; note that this is a random variable.
- Let $\Phi(s)$ be the function that $\mathcal{A}$ is supposed to compute.

An (ϵ, δ)-approximation

- Let $\mathcal{A}(s)$ denote the output of a randomized streaming algorithm $\mathcal{A}$ on input s; note that this is a random variable.
- Let $\Phi(s)$ be the function that $\mathcal{A}$ is supposed to compute.
- $\mathcal{A}$ is a (ϵ, δ)-approximation to Φ if we have

$$Pr\left[\left|\frac{\mathcal{A}(s)}{\Phi(s)} - 1\right| > \epsilon\right] \leq \delta.$$

An (ϵ, δ)-approximation

- Let $\mathcal{A}(s)$ denote the output of a randomized streaming algorithm $\mathcal{A}$ on input s; note that this is a random variable.
- Let $\Phi(s)$ be the function that $\mathcal{A}$ is supposed to compute.
- $\mathcal{A}$ is a (ϵ, δ)-approximation to Φ if we have

$$Pr\left[\left|\frac{\mathcal{A}(s)}{\Phi(s)} - 1\right| > \epsilon\right] \leq \delta.$$

- $\mathcal{A}$ is a (ϵ, δ)-additive approximation to Φ if we have

$$Pr\left[\left|\mathcal{A}(s) - \Phi(s)\right| > \epsilon\right] \leq \delta.$$

An (ϵ, δ)-approximation

- Let $\mathcal{A}(s)$ denote the output of a randomized streaming algorithm $\mathcal{A}$ on input s; note that this is a random variable.
- Let $\Phi(s)$ be the function that $\mathcal{A}$ is supposed to compute.
- $\mathcal{A}$ is a (ϵ, δ)-approximation to Φ if we have

$$Pr\left[\left|\frac{\mathcal{A}(s)}{\Phi(s)} - 1\right| > \epsilon\right] \leq \delta.$$

- $\mathcal{A}$ is a (ϵ, δ)-additive approximation to Φ if we have

$$Pr\left[|\mathcal{A}(s) - \Phi(s)| > \epsilon\right] \leq \delta.$$

- When $\delta = 0$, $\mathcal{A}$ must be deterministic.
  When $\epsilon = 0$, $\mathcal{A}$ must be an exact algorithm.

Counting the number of distinct elements

Counting the number of distinct elements

- For an integer $p > 0$, let zeros(p) be the number of zeros at the end of the binary representation of p.

Counting the number of distinct elements

- For an integer $p > 0$, let $\mathrm{zeros}(p)$ be the number of zeros at the end of the binary representation of p.

$$\mathrm{zeros}(p) = \max\{i \mid 2^i \text{ divides } p\}.$$

Counting the number of distinct elements

- For an integer $p > 0$, let zeros(p) be the number of zeros at the end of the binary representation of p.

$$\text{zeros}(p) = \max\{i \mid 2^i \text{ divides } p\}.$$

- Algorithm:

```
1:  Count-Dif(stream s)
2:  Choose a random hash function h : [n] → [n] form a universal
    family
3:  int z = 0
4:  while not s.end() do
5:     j = s.read()
6:     if zeros(h(j)) > z then
7:        z = zeros(h(j))
8:     end if
9:  end while
10: Return 2^(z+1/2)
```

Counting the number of distinct elements

- For an integer $p > 0$, let zeros(p) be the number of zeros at the end of the binary representation of p.

$$\text{zeros}(p) = \max\{i \mid 2^i \text{ divides } p\}.$$

- Algorithm:

```
1:  Count-Dif(stream s)
2:  Choose a random hash function h : [n] → [n] form a universal
    family
3:  int z = 0
4:  while not s.end() do
5:     j = s.read()
6:     if zeros(h(j)) > z then
7:        z = zeros(h(j))
8:     end if
9:  end while
10: Return 2^(z+1/2)
```

- Assuming that there are d distinct elements, the algorithm computes $\max \text{zeros}(h(j))$ as a good approximation of $\log d$.

Counting the number of distinct elements: Quality

- 1 pass, $O(\log n + \log\log n)$ memory and $O(1)$ time per item.

Counting the number of distinct elements: Quality

- 1 pass, $O(\log n + \log\log n)$ memory and $O(1)$ time per item.
- For $j \in [n]$ and $r \geq 0$, let $X_{r,j}$ be the indicator r.v. for

  $$\text{zeros}(h(j)) \geq r.$$

- Let $Y_r = \sum_{j | f_j > 0} X_{r,j}$.
- Let t denote the final value of z.

Counting the number of distinct elements: Quality

- 1 pass, $O(\log n + \log\log n)$ memory and $O(1)$ time per item.
- For $j \in [n]$ and $r \geq 0$, let $X_{r,j}$ be the indicator r.v. for
  $$\text{zeros}(h(j)) \geq r.$$
- Let $Y_r = \sum_{j|f_j>0} X_{r,j}$.
- Let t denote the final value of z.
- $Y_r > 0$ iff $t \geq r$, or equivalently $Y_r = 0$ iff $t \leq r-1$.

Counting the number of distinct elements: Quality

- 1 pass, $O(\log n + \log\log n)$ memory and $O(1)$ time per item.
- For $j \in [n]$ and $r \geq 0$, let $X_{r,j}$ be the indicator r.v. for $\text{zeros}(h(j)) \geq r$.
- Let $Y_r = \sum_{j|f_j>0} X_{r,j}$.
- Let t denote the final value of z.
- $Y_r > 0$ iff $t \geq r$, or equivalently $Y_r = 0$ iff $t \leq r-1$.
- Since $h(j)$ is uniformly distributed over the $\log n$-bit strings,

$$E[X_{r,j}] = Pr[\text{zeros}(h(j)) \geq r] = Pr[2^r \text{ divides } h(j)] = \frac{1}{2^r}$$

Counting the number of distinct elements: Quality

$$E[X_{r,j}] = Pr[\text{zeros}(h(j)) \geq r] = Pr[2^r \text{ divides } h(j)] = \frac{1}{2^r}.$$

Counting the number of distinct elements: Quality

$$E[X_{r,j}] = Pr[\text{zeros}(h(j)) \geq r] = Pr[2^r \text{ divides } h(j)] = \frac{1}{2^r}.$$

$$E[Y_r] = \sum_{j|f_j>0} E[X_{r,j}] = \frac{d}{2^r}$$

Counting the number of distinct elements: Quality

$$E[X_{r,j}] = Pr[\text{zeros}(h(j)) \geq r] = Pr[2^r \text{ divides } h(j)] = \frac{1}{2^r}.$$

$$E[Y_r] = \sum_{j|f_j>0} E[X_{r,j}] = \frac{d}{2^r}$$

- Random variables Y_r are pairwise independent, as they come from a universal hash family.

$$Var[Y_r] = \sum_{j|f_j>0} Var[X_{r,j}] \leq \sum_{j|f_j>0} E[X_{r,j}^2] = \sum_{j|f_j>0} E[X_{r,j}] = \frac{d}{2^r}$$

Counting the number of distinct elements: Quality

- $E[Y_r] = Var[Y_r] = d/2^r$
- Using Markov's and Chebyshev's inequalities,

$$Pr[Y_r > 0] = Pr[Y_r \geq 1] \leq \frac{E[Y_r]}{1} = \frac{d}{2^r}.$$

$$Pr[Y_r = 0] = Pr[|Y_r - E[Y_r]| \geq \frac{d}{2^r}] \leq \frac{Var[Y_r]}{(d/2^r)^2} \leq \frac{2^r}{d}.$$

Counting the number of distinct elements: Quality

- $Pr[Y_r > 0] \leq \frac{d}{2^r}$ and $Pr[Y_r = 0] \leq \frac{2^r}{d}$.

Counting the number of distinct elements: Quality

- $Pr[Y_r > 0] \leq \frac{d}{2^r}$ and $Pr[Y_r = 0] \leq \frac{2^r}{d}$.
- Let $\hat{d}$ be the estimate of d, $\hat{d} = 2^{t+\frac{1}{2}}$.

Counting the number of distinct elements: Quality

- $Pr[Y_r > 0] \leq \frac{d}{2^r}$ and $Pr[Y_r = 0] \leq \frac{2^r}{d}$.
- Let $\hat{d}$ be the estimate of d, $\hat{d} = 2^{t+\frac{1}{2}}$.
- Let a be the smallest integer so that $2^{a+\frac{1}{2}} \geq 3d$,

$$Pr[\hat{d} \geq 3d] = Pr[t \geq a] = Pr[Y_a = 0] \leq \frac{d}{2^a} \leq \frac{\sqrt{2}}{3}.$$

Counting the number of distinct elements: Quality

- $Pr[Y_r > 0] \leq \frac{d}{2^r}$ and $Pr[Y_r = 0] \leq \frac{2^r}{d}$.
- Let $\hat{d}$ be the estimate of d, $\hat{d} = 2^{t+\frac{1}{2}}$.
- Let a be the smallest integer so that $2^{a+\frac{1}{2}} \geq 3d$,

$$Pr[\hat{d} \geq 3d] = Pr[t \geq a] = Pr[Y_a = 0] \leq \frac{d}{2^a} \leq \frac{\sqrt{2}}{3}.$$

- Let b be the largest integer so that $2^{b+\frac{1}{2}} \leq 3d$,

$$Pr[\hat{d} \leq 3d] = Pr[t \leq b] = Pr[Y_{b+1} = 0] \leq \frac{2^{b+1}}{d} \leq \frac{\sqrt{2}}{3}.$$

Counting the number of distinct elements: Quality

- $Pr[\hat{d} \geq 3d] \leq \frac{\sqrt{2}}{3}$ and $Pr[\hat{d} \leq 3d] \leq \frac{\sqrt{2}}{3}$.
- Thus the algorithm provides a $(2, \frac{\sqrt{2}}{3})$-approximation.

Counting the number of distinct elements: Quality

- $Pr[\hat{d} \geq 3d] \leq \frac{\sqrt{2}}{3}$ and $Pr[\hat{d} \leq 3d] \leq \frac{\sqrt{2}}{3}$.
- Thus the algorithm provides a $(2, \frac{\sqrt{2}}{3})$-approximation.
- How to improve the quality of the approximation?

Counting the number of distinct elements: Quality

- $Pr[\hat{d} \geq 3d] \leq \frac{\sqrt{2}}{3}$ and $Pr[\hat{d} \leq 3d] \leq \frac{\sqrt{2}}{3}$.
- Thus the algorithm provides a $(2, \frac{\sqrt{2}}{3})$-approximation.
- How to improve the quality of the approximation?
- Usual technique: run k several independent copies of the algorithm and take the best information from them, in this case,

Counting the number of distinct elements: Quality

- $Pr[\hat{d} \geq 3d] \leq \frac{\sqrt{2}}{3}$ and $Pr[\hat{d} \leq 3d] \leq \frac{\sqrt{2}}{3}$.
- Thus the algorithm provides a $(2, \frac{\sqrt{2}}{3})$-approximation.
- How to improve the quality of the approximation?
- Usual technique: run k several independent copies of the algorithm and take the best information from them, in this case, the median of the k answers.

Counting the number of distinct elements: Quality

- $Pr[\hat{d} \geq 3d] \leq \frac{\sqrt{2}}{3}$ and $Pr[\hat{d} \leq 3d] \leq \frac{\sqrt{2}}{3}$.
- Thus the algorithm provides a $(2, \frac{\sqrt{2}}{3})$-approximation.
- How to improve the quality of the approximation?
- Usual technique: run k several independent copies of the algorithm and take the best information from them, in this case, the median of the k answers.

  If the median exceed $3d$ at least $k/2$ of the runs do.

Counting the number of distinct elements: Quality

- $Pr[\hat{d} \geq 3d] \leq \frac{\sqrt{2}}{3}$ and $Pr[\hat{d} \leq 3d] \leq \frac{\sqrt{2}}{3}$.
- Thus the algorithm provides a $(2, \frac{\sqrt{2}}{3})$-approximation.
- How to improve the quality of the approximation?
- Usual technique: run k several independent copies of the algorithm and take the best information from them, in this case, the median of the k answers.

  If the median exceed $3d$ at least $k/2$ of the runs do.
- By standard Chernoff bounds, the median exceed $3d$ with probability $2^{-\Omega(k)}$ and the median is below $3d$ with probability $2^{-\Omega(k)}$.

Counting the number of distinct elements: Quality

- $Pr[\hat{d} \geq 3d] \leq \frac{\sqrt{2}}{3}$ and $Pr[\hat{d} \leq 3d] \leq \frac{\sqrt{2}}{3}$.
- Thus the algorithm provides a $(2, \frac{\sqrt{2}}{3})$-approximation.
- How to improve the quality of the approximation?
- Usual technique: run k several independent copies of the algorithm and take the best information from them, in this case, the median of the k answers.

  If the median exceed $3d$ at least $k/2$ of the runs do.
- By standard Chernoff bounds, the median exceed $3d$ with probability $2^{-\Omega(k)}$ and the median is below $3d$ with probability $2^{-\Omega(k)}$.
- Choosing $k = \Theta(\log(1/\delta))$, we can make the sum to be at most δ. So we get a $(2, \delta)$-approximation. However, the used memory is now $O(log(1/\delta) \log n)$.