Algorithms for massive data - notes

Nicola Prezza

November 22, 2021

Contents

1 Overview: processing massive data

The goal of this course is to introduce algorithmic techniques for dealing with **massive data**: data so large that it does not fit in the computer's memory. In such cases, **data compression** comes at rescue with two different solutions: **lossy** and **lossless** compression.

With lossy compression, we throw away uninteresting features of our data and keep only information useful for our purposes. The techniques discussed in Section 3 take this concept to its extreme: we will see how to shrink **exponentially** the size of our data while still being able to compute useful information on it. The most important concept here is **sketching**. Using clever randomized algorithms, we will reduce large data sets to few integers (sketches), taking exponentially less space. Generally speaking, sketches carry enough information to allow us estimating (with some bounded error) some properties of the original data sets, for example set cardinalities and distances (with respect to some metric or similarity function). Sketches can be computed off-line in order to reduce the dataset's size and/or speed up the computation of distances (Section 3), or on-line on a data stream (Section 4), where data is thrown away as soon as it arrives (only data sketches are kept). Section 2 is a self-contained recap of all the probability-theory tools we will need in order to analyze our randomized algorithms (mainly concentration bounds and hashing).

Lossless compression, on the other hand, allows retrieving the original data. In the second part of the course we will exploit the fact that, in some scenarios, data is extremely redundant and can thus be considerably reduced in size (even by orders of magnitude) without losing any information. Interestingly, we will see that often computation on compressed data can be performed **faster** than on the original (uncompressed) data: the active field of **compressed data structures** exploits the natural duality between compression and computation to achieve this feat. The main results we will discuss, **compressed dictionaries and text indexes**, find important applications in information retrieval. They allow to pre-process a large text (or collection of documents) into a compressed data structure (an index) that allows locating substrings very quickly, without decompressing the data. These techniques today stand at the core of modern search engines and sequence-mapping algorithms in computational biology.

Material The proofs of these notes have been reconstructed using various online sources whose links are reported at the end of each section. For other material see the book *Jure Leskovec, Anand Rajaraman, Jeffrey Ullman. Mining of Massive Datasets. 3rd edition.*

2 Probability theory recap

2.1 Random variables

A random variable (R.V.) X is a variable that takes values from some sample space Ω according to the outcomes of a random phenomenon. Ω is also called the *support* of X. Said otherwise, X takes values in Ω according to some probability distribution. A random variable can be discrete if $|\Omega|$ is countable (examples: coin tosses or integer numbers), or continuous (for example, if it takes any real value in some interval). When considering multiple R.V.s with supports $\Omega_1, \ldots, \Omega_n$, the sample space is the Cartesian product of the individual sample spaces: $\Omega = \Omega_1 \times \cdots \times \Omega_n$. For simplicity, in these notes the support of a R.V. will either be a set of integers or an interval of real numbers.

2.1.1 Distribution functions

We indicate with $F(x) = P(X \leq x)$ the *cumulative distribution function of* X: the probability that X takes a value in Ω smaller than or equal to x. $P(X = x) = f(x)$ is the *probability mass function* (for discrete R.V.s) or the *probability density function* (for continuous R.V.s). For discrete R.V.s, this is the probability that X takes value x. For continuous R.V.s, it's the function satisfying $F(x) = \int_{-\infty}^{x} f(x)\, dx$.

Example 2.1. *Take the example of fair coin tosses. Then, $X \in \{0, 1\}$ (0=tail, 1=head) is a discrete random variable with probability mass function $P(X = 0) = P(X = 1) = 0.5$.*

2.1.2 Events

An *event* is a subset of the sample space, i.e. a set of assignments for all the R.V.s under consideration. Each event has a probability to happen. For example, $A = \{0 \leq X \leq 1\}$ is the event indicating that X takes a value between 0 and 1. $P(A \cup B)$ is the probability that either A or B happens. $P(A \cap B)$ is the probability that both A and B happen. Sometimes we will also use the symbols $\vee$ and $\wedge$ in place of $\cup$ and $\cap$ (with the same meaning). $P(A|B)$ indicates the probability that A happens, provided that B has already happened. In general, we have:

$$P(A \cap B) = P(A) \cdot P(B|A)$$

We say that two events A and B are *independent* if $P(A \cap B) = P(A) \cdot P(B)$ or, equivalently, that $P(A|B) = P(A)$ and $P(B|A) = P(B)$: the probability that both happen simultaneously is the product of the probabilities that they happen individually. Said otherwise, the fact that one of the two events has happened, does not influence the happening of the other event.

Example 2.2. *Consider throwing two fair coins, and indicate $A = \{first\ coin\ heads\}$ and $B = \{second\ coin\ heads\}$. The two events are clearly independent, so*

$$P(A \cap B) = P(A) \cdot P(B) = 0.5 \cdot 0.5 = 0.25$$

On the other hand, consider throwing a coin in front of a mirror, and the two events $A = \{coin\ heads\}$ and $B = \{coin\ in\ mirror\ heads\}$. We still have $P(A) = P(B) = 0.5$ (the events, considered separately, have both probability 0.5 to happen), but the two events are clearly dependent! in fact, $P(B|A) = 1 \neq P(B) = 0.5$. So: $P(A \cap B) = P(A) \cdot P(B|A) = P(A) \cdot 1 = 0.5$.

We will often deal with dependent random variables. A useful bound that we will use is the following:

Lemma 2.3 (Union bound)**.** *For any set of (possibly dependent) events $\{A_1, A_2, \ldots, A_n\}$ we have that:*

$$P(\cup_{i=1}^{n} A_i) \leq \sum_{i=1}^{n} P(A_i)$$

The union bound can sometimes give quite uninformative results since the right hand-side sum can exceed 1. The bound becomes extremely useful, however, when dealing with *rare* events: in this case, the probability on the right hand-side could be much smaller than 1. This will be indeed the case in some of our applications.

We finally mention the law of total probability:

Lemma 2.4 (Law of total probability)**.** *If B_i for $i = 1, \ldots, k$ are a partition of the sample space, then for any event A:*

$$P(A) = \sum_{i=1}^{k} P(A \cap B_i) = \sum_{i=1}^{k} P(B_i)P(A|B_i)$$

2.1.3 Expected value and variance

Intuitively, the *expected value* (or mean) $E[X]$ of a numeric random variable X is the arithmetic mean of a large number of independent realizations of X. Formally, it is defined as $E[X] = \sum_{x \in \Omega} x \cdot f(x)$ for discrete R.V.s and $E[X] = \int_{-\infty}^{+\infty} x \cdot f(x)\, dx$ for continuous R.V.s.

Some useful properties of the expected value that we will use:

Lemma 2.5 (Linearity of expectation)**.**

- $E[X + Y] = E[X] + E[Y]$
- $E[a \cdot X] = a \cdot E[X]$*, where a is a constant.*

On the other hand, be aware that in general $E[X \cdot Y] \neq E[X] \cdot E[Y]$. Equality holds if X and Y are independent, though:

$$\begin{aligned} E[XY] &= \textstyle\sum_{i,j} x_i y_j P(X = x_i \wedge Y = y_j) \\ &= \textstyle\sum_{i,j} x_i y_j P(X = x_i) P(Y = y_j) \\ &= \left(\textstyle\sum_i x_i P(X = x_i)\right) \cdot \left(\textstyle\sum_j y_j P(Y = y_j)\right) \\ &= E[X]E[Y] \end{aligned}$$

Also, in general $E[1/X] \neq 1/E[X]$.

The expected value of a non-negative R.V. can also be expressed as a function of the cumulative distribution function. We prove the following equality in the continuous case (the discrete case is analogous), which will turn out useful later in these notes.

Lemma 2.6. *For a non-negative continuous random variable X, it holds:*

$$E[X] = \int_0^\infty P(X \geq x)\, dx$$

Proof. First, express $P(X \geq x) = \int_x^\infty f(t)\, dt$:

$$\int_0^\infty P(X \geq x)\, dx = \int_0^\infty \int_x^\infty f(t)\, dt\, dx$$

In the latter integral, for a particular value of t the value $f(t)$ is included in the summation for *every* value of $x \leq t$. This observation allows us to invert the order of the two integrals as follows:

$$\int_0^\infty \int_x^\infty f(t)\, dt\, dx = \int_0^\infty \int_0^t f(t)\, dx\, dt$$

To conclude, observe that $\int_0^t f(t)\,dx = f(t) \cdot \int_0^t 1\,dx = f(t) \cdot t$, so the latter becomes:

$$\int_0^\infty \int_0^t f(t)\,dx\,dt = \int_0^\infty t \cdot f(t)\,dt = E[X]$$

□

The *Variance* of a R.V. X tells us how much the R.V. deviates from its mean: $Var[X] = E[(X - E[X])^2]$. The following equality will turn out useful:

Lemma 2.7. $Var[X] = E[X^2] - E[X]^2$

Proof. From linearity of expectation: $Var[X] = E[(X - E[X])^2] = E[X^2 - 2XE[X] + E[X]^2] = E[X^2] - 2E[X] \cdot E[E[X]] + E[E[X]^2]$. If Y is a R.V., note that $E[Y]$ is a constant (or, a random variable taking one value with probability 1). The expected value of a constant is the constant itself, thus the above is equal to $E[X^2] - E[X]^2$. □

If X and Y are independent, then one can verify that $Var[X + Y] = Var[X] + Var[Y]$. More in general,

Lemma 2.8. *If $X_1, \ldots, X_n$ are pairwise independent, then $Var[\sum_{i=1}^n X_i] = \sum_{i=1}^n Var[X_i]$.*

Proof. We have $Var[\sum_{i=1}^n X_i] = E[(\sum_{i=1}^n X_i)^2] - E[\sum_{i=1}^n X_i]^2$. The first term evaluates to

$$E[(\sum_{i=1}^n X_i)^2] = \sum_{i=1}^n E[X_i^2] + 2\sum_{i \neq j} E[X_i X_j]$$

Recalling that $E[X_i]E[X_j] = E[X_i X_j]$ if X_i and X_j are independent, the second term evaluates to:

$$E[\sum_{i=1}^n X_i]^2 = \left(\sum_{i=1}^n E[X_i]\right)^2 = \sum_{i=1}^n E[X_i]^2 + 2\sum_{i \neq j} E[X_i]E[X_j] = \sum_{i=1}^n E[X_i]^2 + 2\sum_{i \neq j} E[X_i X_j]$$

Thus, the difference between the two terms is equal to

$$\sum_{i=1}^n E[X_i^2] - \sum_{i=1}^n E[X_i]^2 = \sum_{i=1}^n \left(E[X_i^2] - E[X_i]^2\right) = \sum_{i=1}^n Var[X_i]$$

□

Crucially, note that the above proof does not require full independence (just pairwise independence). This will be important later.

The law of total probability (Lemma 2.4) implies (prove it as an exercise):

Lemma 2.9 (Law of total expectation)**.** *If B_i for $i = 1, \ldots, k$ are a partition of the sample space, then for any random variable X:*

$$E[X] = \sum_{i=1}^k P(B_i)E[X|B_i]$$

2.1.4 Bernoullian R.V.s

Bernoullian R.V.s model the event of flipping a (possibly biased) coin:

Definition 2.10. *A Bernoullian R.V. X takes the value 1 with some probability p (parameter of the Bernoullian), and value 0 with probability $1-p$. The notation $X \sim Be(p)$ means that X is Bernoullian with parameter p.*

Lemma 2.11. *If $X \sim Be(p)$, then X has expected value $E[X] = p$ and variance $Var[X] = p(1-p)$.*

Proof. Note that $X^2 = X$ since $X \in \{0, 1\}$. $E[X] = 0 \cdot (1-p) + 1 \cdot p = p$. $Var[X] = E[X^2] - E[X]^2 = p - p^2 = p(1-p)$. □

Note, as a corollary of the previous lemma, that $Var[X] \leq E[X]$ and $Var[X] \leq 1 - E[X]$ for Bernoullian R.V.s.

2.2 Concentration inequalities

Concentration inequalities provide bounds on how likely it is that a random variable deviates from some value (typically, its expected value). These will be useful in the next sections to calculate the probability of obtaining a good enough approximation with our randomized algorithms.

2.2.1 Markov's inequality

Suppose we know the mean $E[X]$ of a nonnegative R.V. X. Markov's inequality can be used to bound the probability that a random variable takes a value larger than some positive constant. It goes as follows:

Lemma 2.12 (Markov's inequality)**.** *For any nonnegative R.V. X and any $a > 0$ we have:*

$$P(X \geq a) \leq E[X]/a$$

Proof. We prove the inequality for discrete R.V.s (the continuous case is similar). $E[X] = \sum_{x=0}^{\infty} x \cdot P(X = x) \geq \sum_{x=a}^{\infty} x \cdot P(X = x) \geq a \cdot \sum_{x=a}^{\infty} P(X = x) = a \cdot P(X \geq a)$. □

2.2.2 Chebyshev's inequality

Chebyshev's inequality gives us a stronger bound than Markov's, provided that we know the random variable's variance. This inequality bounds the probability that the R.V. deviates from its mean by some fixed value.

Lemma 2.13 (Chebyshev's inequality)**.** *For any $k > 0$:*

$$P(|X - E[X]| \geq k) \leq Var[X]/k^2$$

Proof. We simply apply Markov to to the R.V. $(X - E[X])^2$:

$$P(|X - E[X]| \geq k) = P((X - E[X])^2 \geq k^2) \leq E[(X - E[X])^2]/k^2 = Var[X]/k^2$$

□

Boosting by averaging A trick to get a better bound is to draw s independent values of the R.V. X and average out the results. Let $\hat{X} = \sum_{i=1}^{s} X_i/s$ (also a R.V.) be the average of the s realizations of X. Then:

Lemma 2.14 (Boosted Chebyshev's inequality)**.** *For any $k > 0$ and integer $s \geq 1$:*

$$P(|\hat{X} - E[X]| \geq k) \leq \frac{Var[X]}{s \cdot k^2}$$

Proof. Since the X_i are i.i.d, we have $E[\sum_{i=1}^{s} X_i] = s \cdot E[X]$. For the same reason, $Var[\sum_{i=1}^{s} X_i] = s \cdot Var[X]$. Then:

$$\begin{aligned} P(|\hat{X} - E[X]| \geq k) &= P(s|\hat{X} - E[X]| \geq s \cdot k) \\ &= P(|s\hat{X} - sE[X]| \geq s \cdot k) \\ &= P(|\sum_{i=1}^{s} X_i - E[\sum_{i=1}^{s} X_i]| \geq s \cdot k) \\ &\leq Var[\sum_{i=1}^{s} X_i]/(s \cdot k)^2 \\ &= s \cdot Var[X]/(s \cdot k)^2 \\ &= Var[X]/(s \cdot k^2) \end{aligned}$$

□

2.2.3 Chernoff-Hoeffding's inequalities

Chernoff-Hoeffding's inequalities are used to bound the probability that the sum $Y = \sum_{i=1}^{n} Y_i$ of n independent identically distributed (iid) R.V.s Y_i exceeds by a given value its expectation. The inequalities come in two flavors: with additive error and with relative (multiplicative) error. We will prove both for completeness, but only use the former in these notes. The inequalities give a much stronger bound w.r.t. Markov precisely because we know a particular property of the R.V. Y (i.e. it is a sum of iid. R.V.'s). Here we study the simplified case of Bernoullian R.V.s.

Lemma 2.15 (Chernoff-Hoeffding bound, additive form)**.** *Let $Y_1, \ldots, Y_n$ be independent $Be(p)$ random variables. Denote $Y = \sum_{i=1}^{n} Y_i$. Then, for all $t \geq 0$:*

- $P(Y \geq E[Y] + t) \leq e^{\frac{-t^2}{2n}}$ *[one sided, right]*
- $P(Y \leq E[Y] - t) \leq e^{\frac{-t^2}{2n}}$ *[one sided, left]*
- $P(|Y - E[Y]| \geq t) \leq 2e^{\frac{-t^2}{2n}}$ *[double sided]*

Equivalently, let $\hat{Y} = Y/n$ (i.e. the average of all Y_i) be an estimator for p. Then, for all $0 < \epsilon < 1$:

$$P(|\hat{Y} - p| \geq \epsilon) \leq 2e^{-\epsilon^2 n/2}$$

Proof. Let $X_i = Y_i - E[Y_i]$. Each X_i is distributed in the interval $[-1, 1]$, has mean $E[X_i] = E[Y_i - E[Y_i]] = E[Y_i] - E[Y_i] = 0$, and takes value $1 - E[Y_i] = 1 - p$ with probability p and value $0 - E[Y_i] = -p$ with probability $1 - p$. Let $X = \sum_{i=1}^{n} X_i$. In particular, $X = Y - E[Y]$. We prove $P(X \geq t) \leq e^{\frac{-t^2}{2n}}$. The same argument will hold for $P(-X \geq t) \leq e^{\frac{-t^2}{2n}}$, so by union bound we will get $P(|Y - E[Y]| \geq t) = P(|X| \geq t) \leq 2e^{\frac{-t^2}{2n}}$.

Let $s > 0$ be some free parameter that we will later fix to optimize our bound. The event $X \geq t$ is equivalent to the event $e^{sX} \geq e^{st}$, so:

$$P(X \geq t) = P(e^{s\sum_{i=1}^{n} X_i} \geq e^{st})$$

We apply Markov to the (non-negative[1]) R.V. $e^{s\sum_{i=1}^{n} X_i}$, obtaining

$$P(X \geq t) \leq E[e^{s\sum_{i=1}^{n} X_i}]/e^{st} = E\left[\prod_{i=1}^{n} e^{sX_i}\right]/e^{st}$$

[1]Note: X might be negative, but e^{sX} is always positive, so we can indeed apply Markov's inequality

which, since the X_i's are independent and identically distributed (in particular, they have the same expected value), yields the inequality

$$P(X \geq t) \leq \left(E[e^{sX_1}]\right)^n / e^{st} \tag{1}$$

The goal is now to bound the expected value $E[e^{sX_1}]$ appearing in the above quantity. Define $A = \frac{1+X_1}{2}$ and $B = \frac{1-X_1}{2}$. Note that:

- $A \geq 0$ and $B \geq 0$
- $A + B = \frac{(1+X_1)+(1-X_1)}{2} = 1$
- $A - B = \frac{(1+X_1)-(1-X_1)}{2} = X_1$

We recall Jensen's inequality: if $f(x)$ is convex, then for any $0 \leq a \leq 1$ we have $f(ax + (1-a)y) \leq af(x) + (1-a)f(y)$. Note that e^x is convex so, by the above three observations:

$$\begin{aligned} e^{sX_1} &= e^{s(A-B)} \\ &= e^{sA-sB} \\ &\leq Ae^s + Be^{-s} \\ &= \tfrac{1+X_1}{2}e^s + \tfrac{1-X_1}{2}e^{-s} \\ &= \tfrac{e^s+e^{-s}}{2} + X_1 \cdot \tfrac{e^s-e^{-s}}{2} \end{aligned}$$

Since $E[X_1] = 0$, we obtain:

$$\begin{aligned} E[e^{sX_1}] &\leq E\left[\tfrac{e^s+e^{-s}}{2} + X_1 \cdot \tfrac{e^s-e^{-s}}{2}\right] \\ &= \tfrac{e^s+e^{-s}}{2} + \tfrac{e^s-e^{-s}}{2} \cdot E\left[X_1\right] \\ &= (e^s + e^{-s})/2 \end{aligned}$$

The Taylor expansion of e^s is $e^s = 1 + s + \frac{s^2}{2!} + \frac{s^3}{3!} + \ldots$, while that of e^{-s} is $e^{-s} = 1 - s + \frac{s^2}{2!} - \frac{s^3}{3!} + \ldots$ (i.e. odd terms appear with negative sign). Let *even* and *odd* denote the sum of even and odd terms, respectively. Replacing the two Taylor series in the quantity $(e^s + e^{-s})/2$, we obtain

$$\begin{aligned} (e^s + e^{-s})/2 &= (even + odd)/2 + (even - odd)/2 \\ &= even \\ &= \textstyle\sum_{i=0,2,4,\ldots} \frac{s^i}{i!} \\ &= \textstyle\sum_{i=0}^{\infty} \frac{s^{2i}}{(2i)!} \end{aligned}$$

Now, note that $(2i)! = 1 \cdot 2 \cdot 3 \cdots i \cdot (i+1) \cdots 2i \geq i! \cdot 2^i$, so

$$E[e^{sX_1}] \leq \sum_{i=0}^{\infty} \frac{s^{2i}}{(2i)!} \leq \sum_{i=0}^{\infty} \frac{s^{2i}}{i! \cdot 2^i} = \sum_{i=0}^{\infty} \frac{(s^2/2)^i}{i!}$$

The term $\sum_{i=0}^{\infty} \frac{(s^2/2)^i}{i!}$ is precisely the Taylor expansion of $e^{s^2/2}$. We conclude

$$E[e^{sX_1}] \leq e^{s^2/2}$$

and Inequality 1 becomes

$$P(X \geq t) \leq \left(e^{s^2/2}\right)^n / e^{st} = e^{(ns^2 - 2st)/2} \tag{2}$$

Recall that s is a free parameter. In order to obtain the strongest bound, we have to minimize $e^{(ns^2-2st)/2}$ as a function of s. This is equivalent to minimizing $ns^2 - 2st$. The coefficient of the second-order term is $n > 0$, so the polynomial indeed has a minimum. In order to find it, we find the root of its derivative: $2ns - 2t = 0$, which tells us that the minimum occurs at $s = t/n$. Replacing $s = t/n$ in Inequality 2, we finally obtain $P(X \geq t) \leq e^{-t^2/(2n)}$. □

If $E[Y]$ is small, a bound on the relative error is often more useful:

Lemma 2.16 (Chernoff-Hoeffding bound, multiplicative form)**.** *Let $Y_1, \ldots, Y_n$ be independent $Be(p)$ random variables. Denote $Y = \sum_{i=1}^{n} Y_i$ and $\mu = E[Y] = np$. Then, for all $0 < \epsilon < 1$:*

- $P(Y \geq (1+\epsilon)\mu) \leq e^{-\mu\epsilon^2/3}$ *[one sided, right]*
- $P(Y \leq (1-\epsilon)\mu) \leq e^{-\mu\epsilon^2/2}$ *[one sided, left]*
- $P(|Y - \mu| \geq \epsilon\mu) \leq 2e^{-\epsilon^2\mu/3}$ *[double sided]*

Proof. We first study $P(Y \geq (1+\epsilon)\mu)$. Note that $\mu = np$, since our R.V.s are distributed as $Be(p)$.

The first step is to upper-bound the quantity $P(Y \geq t)$ (later we will fix $t = (1+\epsilon)\mu$). Let $s > 0$ be some parameter that we will later fix to optimize our bound. The event $Y \geq t$ is equivalent to the event $e^{sY} \geq e^{st}$, so:

$$P(Y \geq t) = P(e^{s\sum_{i=1}^{n} Y_i} \geq e^{st})$$

We apply Markov to the R.V. $e^{s\sum_{i=1}^{n} Y_i}$, obtaining

$$P(Y \geq t) \leq E[e^{s\sum_{i=1}^{n} Y_i}]/e^{st} = E\left[\prod_{i=1}^{n} e^{sY_i}\right]/e^{st}$$

which, since the Y_i's are independent and identically distributed (in particular, they have the same expected value), yields the inequality

$$P(Y \geq t) \leq \left(E[e^{sY_1}]\right)^n / e^{st} \tag{3}$$

Replacing $t = (1+\epsilon)\mu$, we obtain

$$P(Y \geq (1+\epsilon)\mu) \leq \left(E[e^{sY_1}]\right)^n / e^{s(1+\epsilon)\mu} = \left(E[e^{sY_1}]e^{-sp(1+\epsilon)}\right)^n \tag{4}$$

The expected value $E[e^{sY_1}]$ can be bounded as follows:

$$\begin{aligned} E[e^{sY_1}] &= p \cdot e^{s\cdot 1} + (1-p)\cdot e^{s\cdot 0} \\ &= p \cdot e^s + 1 - p \\ &= 1 + p(e^s - 1) \\ &\leq e^{p(e^s-1)} \end{aligned}$$

where in the last step we used the inequality $1 + x \leq e^x$ with $x = p(e^s - 1)$. Combining this with Inequality 4 we obtain:

$$P(Y \geq (1+\epsilon)\mu) \leq \left(e^{p(e^s-1)}e^{-sp(1+\epsilon)}\right)^n = \left(e^{e^s-1}e^{-s(1+\epsilon)}\right)^\mu \tag{5}$$

By taking $s = \log(1+\epsilon)$ (it can be shown that this choice optimizes the bound), we have $e^{e^s-1}e^{-s(1+\epsilon)} = e^{\epsilon - \log(1+\epsilon)^{(1+\epsilon)}}$, thus Inequality 5 becomes:

$$P(Y \geq (1+\epsilon)\mu) \leq \left(\frac{e^\epsilon}{(1+\epsilon)^{(1+\epsilon)}}\right)^\mu = \rho \tag{6}$$

To conclude, we bound $\log \rho = \mu(\epsilon - (1+\epsilon)\log(1+\epsilon))$. We use the inequality $\log(1+\epsilon) \geq \frac{\epsilon}{1+\epsilon/2}$, which holds for all $\epsilon \geq 0$, and obtain:

$$\log \rho \leq \mu \left(\epsilon - \frac{\epsilon(1+\epsilon)}{1+\epsilon/2} \right) = \frac{-\mu\epsilon^2}{2+\epsilon} \leq \frac{-\mu\epsilon^2}{3} \tag{7}$$

Where the latter inequality holds since we assume $\epsilon < 1$. Finally, Bounds 6 and 7 yield:

$$P(Y \geq (1+\epsilon)\mu) \leq e^{-\mu\epsilon^2/3} \tag{8}$$

We are left to find a bound for the symmetric tail $P(Y \leq (1-\epsilon)\mu) = P(-Y \geq -(1-\epsilon)\mu)$. Following the same procedure used to obtain Inequality 4 we have

$$P(-Y \geq -(1-\epsilon)\mu) \leq \left(E[e^{-sY_1}] e^{s(1-\epsilon)p} \right)^n$$

We can bound the expectation as follows: $E[e^{-sY_1}] = p \cdot e^{-s} + (1-p) = 1 + p(e^{-s} - 1) \leq e^{p(e^{-s}-1)}$ and obtain:

$$P(-Y \geq -(1-\epsilon)\mu) \leq \left(e^{e^{-s}-1} e^{s(1-\epsilon)} \right)^\mu \tag{9}$$

It can be shown that the bound is minimized for $s = -\log(1-\epsilon)$. This yields:

$$P(Y \leq (1-\epsilon)\mu) \leq \left(\frac{e^{-\epsilon}}{(1-\epsilon)^{(1-\epsilon)}} \right)^\mu = \rho \tag{10}$$

Then, $\log \rho = \mu(-\epsilon - (1-\epsilon)\log(1-\epsilon))$. We plug the bound $\log(1-\epsilon) \geq \frac{\epsilon^2/2 - \epsilon}{1-\epsilon}$, which holds for all $0 \leq \epsilon < 1$. Then, $\log \rho \leq \mu(-\epsilon - (\epsilon^2/2 - \epsilon)) = -\mu\epsilon^2/2$. This yields

$$P(Y \leq (1-\epsilon)\mu) \leq e^{-\mu\epsilon^2/2} \leq e^{-\mu\epsilon^2/3} \tag{11}$$

and by union bound we obtain our double-sided bound. □

Equivalently, we can bound the probability that the arithmetic mean of n independent R.V.s deviates from its expected value. This yields a useful estimator for Bernoullian R.V.s (i.e. the arithmetic mean of n independent observations of a Bernoullian R.V.). Note that the bound improves *exponentially* with the number n of samples.

Corollary 2.16.1. *Let $Y_1, \ldots, Y_n$ be independent $Be(p)$ random variables. Consider the estimator $\hat{Y} = \frac{1}{n}\sum_{i=1}^n Y_i$ for the value p $(= E[\hat{Y}])$. Then, for all $0 < \epsilon < 1$:*

$$P(|\hat{Y} - p| \geq \epsilon p) \leq 2e^{-\epsilon^2 np/3}$$

2.3 Hashing

A *hash function* is a function $h : U \to [0, M)$ from some universe U (usually, an interval of integers) to an interval of numbers (usually the integers, but we will also work with the reals). Informally speaking, h is used to *randomizes our data* and should have the following basic features:

1. $h(x)$ should be "as random" as possible. Ideally, h should map the elements of U completely uniformly (but we will see that this has a big cost).
2. $h(x)$ should be quick to compute algorithmically. Ideally, we would like to compute $h(x)$ in time proportional to the time needed to read x ($O(1)$ if x is an integer, or $O(n)$ if x is a string of length n).
3. $h(x)$ should take very little space in memory (ideally, $O(1)$ words of space).

$h(x)$ will also be called *the fingerprint* of x.

Note that, while h accepts as input any value from U, typically the algorithms using h will apply it to much smaller subsets of U (for example, U might be the set of all 2^{32} possible IPv4 addresses, but the algorithm will work on just a small subset of them).

We formalize the notion of hashing as follows. We define a family $\mathcal{H} \subseteq [0, M)^{|U|}$ of functions (each function assigns a value from $[0, M)$ to each of the $|U|$ universe elements) and extract a uniform[2] $h \in \mathcal{H}$. Then, we run our algorithm using the chosen h. The expected-case analysis of the algorithm will take into account the structure of $\mathcal{H}$ and the fact that h has been chosen uniformly from it. By a simple information-theoretic argument, it is easy to see that in the worst case we need at least $\log_2 |\mathcal{H}|$ bits in order to represent (and store in memory) h: if, for a contradiction, we used $t < \log_2 |\mathcal{H}|$ bits for all $h \in \mathcal{H}$, then we would be able to distinguish only among at most $2^t < |\mathcal{H}|$ functions, which is not sufficient since (by uniformity of our choice) any h could be chosen from the set.

Ideally, we would like our hash function to be uniform:

Definition 2.17 (Uniform hash function)**.** *Assume that $h \in \mathcal{H} \subseteq [0, M)^{|U|}$ is chosen uniformly. We say that $\mathcal{H}$ is* uniform *if for any $x_1, \ldots, x_{|U|} \in [0, M)$, we have $P(h = (x_1, \ldots, x_{|U|})) = \frac{1}{M^{|U|}}$.*

As it turns out, the requirements (1-3) above are in conflict: in fact, it is impossible to obtain all three simultaneously. Assume, for example, that our goal is to obtain a uniform hash function. Then, for any choice of $x_1, \ldots, x_{|U|} \in [0, M)$, $P(h = (x_1, \ldots, x_{|U|})) = \frac{1}{M^{|U|}}$. However, this is possible only if $\mathcal{H} = [0, M)^{|U|}$: in any other case, there would exist at least one choice of $x_1, \ldots, x_{|U|} \in [0, M)$ such that $P(h = (x_1, \ldots, x_{|U|})) < \frac{1}{M^{|U|}}$. Therefore, a uniform hash function must take $\log_2 |\mathcal{H}| = \log_2 M^{|U|} = |U| \log_2 M$ bits of memory, which is typically too much. It is easy to devise such a hash function: fill a vector $V[1, |U|]$ with uniform integers from $[0, M)$, and define $h(x) = V[x]$. Note that such a hash function satisfies requirements (1) and (2), but not (3). In the next subsection we study good compromises that will work for many algorithms: k-uniform (or k-wise independent) and universal hashing. Then, we briefly discuss functions mapping U to real numbers.

2.3.1 k-uniform/universal hashing

In this section we work with integer hash functions: $h : [1, n] \to [0, M)$.

k-uniform (or k-independent or k-wise independent) hashing is a weaker version of uniform hashing:

Definition 2.18. *We say that the family $\mathcal{H}$ is k-uniform (or k-independent or k-wise independent) if and only if, for a uniform choice of $h \in \mathcal{H}$, we have that*

$$P\left(\bigwedge_{i=1}^{k} h(x_i) = y_i\right) = M^{-k}$$

for any choice of distinct $x_1, \ldots, x_k \in [1, n]$ and (not necessarily distinct) $y_1, \ldots, y_k \in [0, M)$.

Thus, a uniform $\mathcal{H}$ is the particular case where $k = n$. In other words, h maps k-tuples uniformly: the k-tuple $(h(x_1), \ldots, h(x_k))$ is a uniform random variable over $[0, M)^k$ when $x_1, \ldots, x_k$ are distinct. In the next sections we will see that $k = 2$ is already sufficient in many interesting cases: this case is also called *two-independent hashing*.

Another important concept is that of *universality*:

Definition 2.19. *We say that $\mathcal{H}$ is universal if and only if, for a uniform choice of $h \in \mathcal{H}$, we have that*

$$P\left(h(x_1) = h(x_2)\right) \leq 1/M$$

for any choice of distinct $x_1 \neq x_2 \in [1, n]$.

[2] One (strong) assumption is always needed for this to work: we can draw uniform integers. This is actually impossible, since computers are deterministic. However, there is a vast literature on pseudo-random number generators (PRNG) which behave reasonably well in practice. We will thus ignore this problem for simplicity.

Note that this is at most the probability of collision we would expect if the hash function assigned truly random outputs to every key. It is easy to see that two-independence implies universality. Consider the partition of the sample space $\{h(x_2) = y\}_{y\in[0,M)}$. By the law of total probability (Lemma 2.4):

$$\begin{aligned} P\left(h(x_1) = h(x_2)\right) &= \textstyle\sum_{y\in[0,M)} P(h(x_1) = h(x_2) \wedge h(x_2) = y) \\ &= \textstyle\sum_{y\in[0,M)} P(h(x_1) = y \ \wedge \ h(x_2) = y) \\ &= \textstyle\sum_{y\in[0,M)} M^{-2} \\ &= 1/M \end{aligned}$$

Next, we show a construction (not the only possible one) yielding a two-independent hash function. Let $M > n$ be a prime number, and define

$$h_{a,b}(x) = (a \cdot x + b) \mod M$$

Our family $\hat{\mathcal{H}}$ as

$$\hat{\mathcal{H}} = \{h_{a,b} \ : \ a, b \in [0, M)\}$$

In other words, a uniform $h_{a,b} \in \hat{\mathcal{H}}$ is a uniformly-random polynomial of degree 1 over $\mathbb{Z}_M$. Note that this function is fully specified by a, b, M and thus it can be stored in $O(\log M)$ bits. Moreover, $h_{a,b}$ can clearly be evaluated in $O(1)$ time. In our applications, the primality requirement for M is not restrictive since for any x, a prime number always exists between x and $2x$ (and, on average, between x and $x + \ln(x)$). This will be enough since we will only require asymptotic guarantees from M (e.g. $M \in \Theta(n^c)$ for some constant c).

We now prove 2-uniformity (a.k.a. two-independence).

Lemma 2.20. *$\hat{\mathcal{H}}$ is a two-independent family.*

Proof. Pick any distinct $x_1, x_2 \in [1, n]$ and (not necessarily distinct) $y_1, y_2 \in [0, M-1)$. Crucially, note that since $x_1 \neq x_2$ and $M > n$, then $x_1 \not\equiv_M x_2$.

$$\begin{aligned} P\left(h(x_1) = y_1 \wedge h(x_2) = y_2\right) &= P(ax_1 + b \equiv_M y_1 \wedge ax_2 + b \equiv_M y_2) \\ &= P\left(b \equiv_M y_1 - x_1 \cdot \tfrac{y_2-y_1}{x_2-x_1} \wedge a \equiv_M \tfrac{y_2-y_1}{x_2-x_1}\right) && (a) \\ &= P\left(b \equiv_M y_1 - x_1 \cdot \tfrac{y_2-y_1}{x_2-x_1}\right) \cdot P\left(a \equiv_M \tfrac{y_2-y_1}{x_2-x_1}\right) && (b) \\ &= M^{-2} \end{aligned}$$

Notes:

(a) Simply solve the system in the variables a and b. Note that $(x_2 - x_1)^{-1}$ exists because $x_2 \not\equiv_M x_1$ and $\mathbb{Z}_M$ is a field, thus every element (except 0) has a multiplicative inverse.

(b) a and b are independent random variables.

□

In general, it can be proved that the family $\hat{\mathcal{H}} = \left\{\sum_{i=0}^{k-1} a_i x^i \mod M \ : \ a_0, \ldots, a_{k-1} \in [0, M)\right\}$ is k-uniform whenever $M > n$ is a power of a prime number. Note that members of this family take $O(k \log M)$ bits of space to be stored and can be evaluated in $O(k)$ time.

For the special case $k = 1$ of k-uniform hashing, we obtain the following definition:

Definition 2.21. *Assume that $h \in \mathcal{H} \subseteq [0, M)^{|U|}$ is chosen uniformly. We say that $\mathcal{H}$ is* 1-uniform *(or just* uniform*) if for any $x \in U$ and $y \in [0, M)$, we have $P(h(x) = y) = 1/M$.*

In other words, a uniform h is not biased towards any value. Pairwise independence implies uniformity. Universality, however, does not imply uniformity.

2.3.2 Collision-free hashing

In some applications we will need a *collision-free* hash function:

Definition 2.22. *A hash function $h : [1, n] \to [0, M)$ is collision-free on a set $A \subseteq [1, n]$ if, for any $x_1 \neq x_2$, $x_1, x_2 \in A$, we have $h(x_1) \neq h(x_2)$.*

In general we will be happy with a function that satisfies this property *with high probability*:

Definition 2.23 (with high probability (w.h.p.))**.** *We say that an event holds with high probability with respect to some quantity n, if its probability is at least $1 - n^{-c}$ for an arbitrarily large constant c. Equivalently, we say that the event succeeds with inverse-polynomial probability.*

To simplify our analyses, in these notes we will ignore the incredibly small failure probability of events holding with high probability, and simply assume that they happen with probability 1. Note that this is reasonable in practice: for example, on a small universe with $n = 10^6$ (typical universes are much larger than that), the small constant $c = 3$ already gives a failure probability of 10^{-18}. It is far more likely that your program fails because a cosmic ray has flipped a bit in RAM [3].

We prove:

Lemma 2.24. *If $h : [1, n] \to [0, M)$ is universal and $M \geq n^{c+2}$ for an arbitrarily large constant c, then h is collision-free on any set $A \subseteq [1, n]$ with high probability, i.e. with probability at least $1 - n^{-c}$.*

Proof. Universality means that $P(h(x_1) = h(x_2)) \leq M^{-1}$ for any $x_1 \neq x_2$. Since there are at most $|A|^2 \leq n^2$ pairs of distinct elements in $|A|$, by union bound the probability of having at least one collision is at most n^2/M. By choosing $M \geq n^{c+2}$ we have at least one collision with probability at most n^{-c}, i.e. our function is collision-free with probability at least $1 - n^{-c}$. □

Note that, by choosing $M \in \Theta(n^{c+2})$, one hash value (as well as the hash function itself) can be stored in $\log_2 M \in O(\log n)$ bits and can be evaluated in constant time.

2.3.3 Hashing integers to the reals

If $h : [1, n] \to [0, 1]$ is a hash function mapping the integers $[1, n]$ to the real interval $[0, 1]$, we say that h is k-uniform iff $(h(x_1), \ldots, h(x_k))$ is uniform in $[0, 1]^k$ for any choice of distinct $x_1, \ldots, x_k$.

It is impossible to algorithmically draw (and store) a uniform function $h : [1, n] \to [0, 1]$, since the interval $[0, 1]$ contains infinitely-many numbers. However, we can aim at an approximation with any desired degree of precision (i.e. decimal digits of $h(x)$). Here we show how to simulate such a two-independent hash function (enough for the purposes of these notes).

We start with a two-independent discrete hash function $h' : [1, n] \to [0, M]$ (just $O(\log M)$ bits of space, see the previous subsection) that maps integers from $[1, n]$ to integers from $[0, M]$ and define $h(x) = h'(x)/M \in [0, 1]$. Since h' is two-independent, also h is two-independent (over our approximation of $[0, 1]^2$).

In addition to being two-independent, h' should be collision-free on the subset of $[1, n]$, of size $d \leq n$, on which the algorithm will work. This is required because, on a truly uniform $h : [1, n] \to [0, 1]$, we have $P(h(x) = h(y)) = 0$ whenever $x \neq y$. We will be happy with a guarantee that holds with high probability. Recalling that two-independence implies universality, by the discussion of the previous section it is sufficient to choose $M \geq n^{c+2}$ to obtain a collision-free hash function.

2.3.4 Hash tables

Since our hash functions $h : [1, n] \to [0, M)$ have good statistical properties (in particular, low collision rate), can we use them as an index in an array $H[0, \ldots, M - 1]$ to implement a *dictionary data structure*? The collision-free property ensures that $H[h(x)]$ is likely to contain only (data associated

[3] `stackoverflow.com/questions/2580933/cosmic-rays-what-is-the-probability-they-will-affect-a-program`

with) x, so it looks like this is going to be a fast data structure with constant-time insert/access/delete operations. Note, however, that we required $M \approx n^c$ for some constant $c > 2$, so the space of H would be prohibitively large.

The problem we want to solve is:

Definition 2.25 (Hash table)**.** *A* hash table *over* $[1, n]$ *is a data structure* H *implementing a set. We want* H *to support quickly the following operations:*

- *Insert* x *in* H
- *Check if* $x \in H$
- *Remove* x *from* H

After inserting d *elements, the space of* H *should be bounded by* $O(d)$ *words.*

We now give an implementation of a hash table: *hashing by chaining*. Assuming we know the number d of elements that will be inserted in our hash table, we choose a hash function $\bar{h} : [1, n] \to [0, d)$ and initialize an empty vector $H[0, d-1]$ of size d (indexed from index 0). The cell $H[i]$ contains an array—we call it the *i-th chain*—, initially of size 0 (to be more precise, $H[i]$ is a pointer to a resizable array). Then, operation *insert* will be implemented by appending x at the end of the chain $H[h(x)]$. Arrays are resized using a doubling technique, so that they occupy linear space and support appending an integer in constant amortized time. Of course, we want the collision probability of $\hat{h}$ to be as low as possible: for any $x \neq y$, we want $P(h(x) = h(y)) \leq 1/d$. We now describe a hash function achieving this property.

Definition 2.26. *Choose a prime number* $M > n^{c+2}$ *for an arbitrarily large constant* c*. Then, choose two uniform numbers* $a, b \in [0, M)$*. Our hash function is:*

$$h(x) = ((a \cdot x + b) \mod M) \mod d$$

Lemma 2.27. *Function* $h(x)$ *is universal, i.e. for any* $x \neq y$*, it holds* $P(h(x) = h(y)) \leq 1/d$*.*

Proof. Let $X = (a \cdot x + b) \mod M$ and $Y = (a \cdot y + b) \mod M$. The equality $h(x) = h(y)$ means that X and Y give the same remainder when divided by d, i.e. $X \equiv_d Y$. From Lemma 2.20, X and Y are independent uniform random variables with support $[0, M)$. Now, fix a given X. In the interval $[0, M)$, there are at most M/d values for Y such that $Y \equiv_d X$: those are all the integers whose distance from X is a multiple of d. It follows that, since Y is chosen uniformly from $[0, M)$, the chance of picking such a value of Y is at most $(M/d)/M = 1/d$. We can finally apply the law of total probability on the partition $X = 0, \ldots, M-1$ of the event space and obtain $P(h(x) = h(y)) = \sum_{r \in [0,M)} P(X = r) \cdot P(Y \equiv_d r) \leq \sum_{r \in [0,M)} (1/M) \cdot (1/d) \leq 1/d$. □

Let $x_1, \ldots, x_d$ be the d elements in the hash table. Universality of h implies that the expected length $|H[h(x_i)]|$ of the chain associated with any element x_i is $E[|H[h(x_i)]|] = E[\sum_{j \neq i} \mathbb{1}_{h(x_i)=h(x_j)}] = \sum_{j \neq i} E[\mathbb{1}_{h(x_i)=h(x_j)}] \leq d \cdot (1/d) = 1$. This implies that each operation on the hash table takes expected constant time.

To conclude, we can lift the assumption that we know d in advance. Suppose we have allocated a hash table of size d. After having inserted the d-th element, we allocate a new table of size $2d$, re-hash all elements in this new table using a new hash function modulo $2d$, and delete the old table. It is easy to see that the total space remains linear and operations still take constant amortized time.

3 Similarity-preserving sketching

Let x be some data: a set, a string, an integer, etc. A *data sketch* is a randomized function f mapping x to a (short) sequence of bits with the following interesting properties:

1. f is easy to compute.
2. $f(x)$ is easy to update if x gets updated (e.g. we add an element if x is a set, or we append a character if x is a string). This may include combining sketches (e.g. to obtain the sketch of the union, if the data represents sets).
3. $f(x)$ can be used to efficiently compute some properties of x (e.g. the number of distinct elements contained in x, if x is a set).

A *similarity-preserving sketch* has a somewhat stronger property that allows comparing sketches: if x and y are similar (according to some measure of similarity, e.g. Euclidean distance), then $f(x)$ and $f(y)$ are likely to be similar (according to some measure of similarity, not necessarily the same as before). Note that $f(x)$ and $f(y)$ are (in general, dependent) random variables, being f a randomized function. Clearly, f becomes interesting if $|f(x)|$ (number of bits needed to represent $f(x)$) is much smaller than $|x|$. In general, we will discuss sketches whose sizes are poly-logarithmic with respect to $|x|$ (the size will also depend on other parameters, such as error rate and probability of obtaining a good approximation).

3.1 Identity - Karp-Rabin hashing

The most straightforward measure of similarity is identity: given x and y, is x equal to y? Without loss of generality, let x be a string over alphabet Σ. Note that this setting can also be used to represent subsets of $[1, n]$, letting $\Sigma = \{0, 1\}$ and $|x| = n$. Observe that, for any function f, if $|f(x)| < |x|$ then collisions must occur: there must exist pairs $x \neq y$ such that $f(x) = f(y)$. *Karp-Rabin hashing* is a powerful technique guaranteeing that collisions are extremely rare events.

Definition 3.1 (Karp-Rabin hash function)**.** *Fix a prime number q, and pick a uniform $z \in [0, q)$. Let $x[1, n] \in \Sigma^n$ be a string of length n. The Karp-Rabin hash function $\kappa_{q,z}(x)$ is defined as:*

$$\kappa_{q,z}(x) = \left(\sum_{i=0}^{n-1} x[n-i] \cdot x^i \right) \mod q$$

In other words: $\kappa_{q,z}(x)$ is a polynomial modulo q evaluated in z (a random point) and having as coefficients the characters of x. [4]

First, we show that $\kappa_{q,z}(x)$ is easy to compute and update. Suppose we wish to append a character c at the end of x, thereby obtaining a string $x \cdot c$. It is easy to see that this can be achieved as follows (Horner's method for evaluating polynomials):

Lemma 3.2. $\kappa_{q,z}(x \cdot c) = (\kappa_{q,z}(x) \cdot z + c) \mod q$

The above lemma gives us an efficient algorithm for computing $\kappa_{q,z}(x)$: start from $\kappa_{q,z}(\epsilon) = 0$ (where ϵ is the empty string) and append the characters of x one by one.

Even better: using a similar idea, we can concatenate the sketches of two strings in logarithmic time. Let us denote with $x \cdot y$ the concatenation of the two string x and y. For this to work, our sketch should also remember the string's length (only an additional logarithmic number of bits): the sketch of x becomes the pair $(\kappa_{q,z}(x), |x|)$, where $|x|$ denotes the number of characters in x. For simplicity, in the following we will omit the string's length (but assume we store it in the sketch). It is easy to see that:

[4] Another variant of Karp-Rabin hashing draws a uniform prime q instead, and fixes $z = |\Sigma|$

Lemma 3.3. $\kappa_{q,z}(x \cdot y) = (\kappa_{q,z}(x) \cdot z^{|y|} + \kappa_{q,z}(y)) \mod q$

The length of the resulting string is, of course, $|x \cdot y| = |x| + |y|$. Even if it appears that the above formula can be computed in constant time, the quantity $z^{|y|} \mod q$ actually requires $O(\log |y|)$ time to be computed (by means of the fast exponentiation algorithm). [5]

We mention an additional crucial property of Karp-Rabin hashing: if $x \neq y$, then $\kappa_{q,z}(x) \neq \kappa_{q,z}(y)$ with high probability. This is implied by the following lemma:

Lemma 3.4. *Let $x \neq y$, with $\max(|x|, |y|) = n$. Then:*

$$P(\kappa_{q,z}(x) = \kappa_{q,z}(y)) \leq n/q$$

Proof. Note that $P(\kappa_{q,z}(x) = \kappa_{q,z}(y)) = P(\kappa_{q,z}(x) - \kappa_{q,z}(y) \equiv_q 0)$. Now, the quantity $\kappa_{q,z}(x) - \kappa_{q,z}(y)$ is, itself, a polynomial. Let $x - y$ be the string such that $(x - y)[i] = x[i] - y[i] \mod q$, where we left-pad with zeros the shortest of the two strings (so that both have n characters). Then, it is easy to see that:

$$\kappa_{q,z}(x) - \kappa_{q,z}(y) \mod q = \kappa_{q,z}(x - y)$$

It follows that the above probability is equal to $P(\kappa_{q,z}(x - y) \equiv_q 0)$. Since $x \neq y$, $\kappa_{q,z}(x - y)$ is a polynomial of degree at most n over $\mathbb{Z}_q$ (evaluated in z) and it is not the zero polynomial. Recall that any univariate polynomial of degree n over a finite field has at most n roots. Since q is prime, $\mathbb{Z}_q$ is a field and thus there are at most n values of z such that $\kappa_{q,z}(x - y) \equiv_q 0$. Since we pick z uniformly from $[0, q)$, the probability of picking a root is at most n/q. □

Corollary 3.4.1. *Choose a prime $n^{c+1} \leq q \leq 2 \cdot n^{c+1}$ for an arbitrarily large constant c. Then, $|\kappa_{q,z}(x)| \in O(\log n)$ bits and, for any $x \neq y$:*

$$P(\kappa_{q,z}(x) = \kappa_{q,z}(y)) \leq n^{-c}$$

that is, x and y collide with low (inverse polynomial) probability.

Later in these notes, Karp-Rabin fingerprinting will be used to solve pattern matching in the streaming model. As noted above, this framework can be used also to sketch sets of integers. Note that, in this case, the sketch can be efficiently updated also when a new element is inserted in the set (provided that the element was not in the set before).

3.2 Jaccard similarity - MinHash

Here we report just a definition and analysis of MinHash. For more details and applications see the book: Jure Leskovec, Anand Rajaraman, Jeffrey Ullman. *Mining of Massive Datasets*, `www.mmds.org`, Sections 3.1 - 3.3.

MinHash is a technique for estimating the Jaccard similarity $J(A, B)$ of two sets A and B:

Definition 3.5 (Jaccard similarity)**.** $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$

Without loss of generality, we may assume that we work with sets of integers from the universe $[1, n]$. This is not too restrictive: for example, if we represent a document as the set of all substrings of some length k appearing in it, we can convert those strings to integers using Karp-Rabin hashing.

Definition 3.6 (MinHash hash function)**.** *Let h be a hash function. The MinHash hash function of a set A is defined as $\hat{h}(A) = \min\{h(x) \ : \ x \in A\}$, i.e. it is the minimum of h over all elements of A.*

[5] An alternative solution is to pre-compute all powers $z^i \mod q$, for $1 \leq i \leq n$. This, however, requires $O(n)$ space. Another clever solution is to store the hash of an overlapping prefix-suffix pair of the string, both of length equal to a power of two. Then, precomputing powers requires just $O(\log n)$ space.

Definition 3.7 (MinHash estimator)**.** *Let $\hat{J}_h(A,B)$ be the indicator R.V. defined as follows:*

$$\hat{J}_h(A,B) = \begin{cases} 1 & \text{if } \hat{h}(A) = \hat{h}(B) \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

Note that $\hat{J}_h(A,B)$ is a Bernoullian R.V. The book proves that, if h is a uniform permutation (i.e. $h : [1,n] \to [1,n]$), then $E[\hat{J}_h(A,B)] = J(A,B)$: the success probability of the random variable $\hat{J}_h(A,B)$ is $J(A,B)$. However, storing a completely uniform permutation requires too much space ($n \log n$ bits). Here we show that the claim still holds for much simpler functions:

Lemma 3.8. *If h is 1-uniform and collision-free on $A \cup B$, then $E[\hat{J}_h(A,B)] = J(A,B)$*

Proof. Let $|A \cup B| = N$. For $i \in A \cup B$, consider the event $smallest(i) = (\forall j \in A \cup B)(h(i) < h(j))$, stating that i is the element mapped to the smallest hash $h(i)$. Since we assume that h is collision-free, exactly one element from $A \cup B$ will be mapped to the smallest hash (i.e. $smallest(i)$ is true for exactly one $i \in A \cup B$), so $\{smallest(i)\}_{i \in A \cup B}$ is a partition of cardinality $N = |A \cup B|$ of the event space. Moreover, the fact that h is 1-uniform implies that $P(smallest(i)) = P(smallest(j))$ for all $i, j \in A \cup B$: every element has the same chance to be mapped to the smallest hash. This implies that $P(smallest(i)) = 1/N$ for every $i \in A \cup B$.

Note that, if we know that $smallest(i)$ is true and $i \in A \cap B$, then $\hat{J}_h(A,B) = 1$ (because i belongs to both A and B and h reaches its minimum `min` on i, thus $\hat{h}(A) = \hat{h}(B) = \texttt{min}$). On the other hand, if we know that $smallest(i)$ is true and $i \in (A \cup B) - (A \cap B)$, then $\hat{J}_h(A,B) = 0$ (because i belongs to either A or B — not both — and h reaches its minimum `min` on i, thus either $\hat{h}(A) \neq \hat{h}(B) = \texttt{min}$ or $\texttt{min} = \hat{h}(A) \neq \hat{h}(B)$ holds).

Using this observation and applying the law of total expectation (Lemma 2.9) to the partition $\{smallest(i)\}_{i \in A \cup B}$ of the event space we obtain:

$$\begin{aligned}
E[\hat{J}_h(A,B)] &= \textstyle\sum_{i \in A \cup B} P(smallest(i)) \cdot E[\hat{J}_h(A,B) \mid smallest(i)] \\
&= \textstyle\sum_{i \in A \cup B} \frac{1}{N} \cdot E[\hat{J}_h(A,B) \mid smallest(i)] \\
&= \textstyle\sum_{i \in A \cap B} \frac{1}{N} \cdot E[\hat{J}_h(A,B) \mid smallest(i)] + \sum_{i \in (A \cup B) - (A \cap B)} \frac{1}{N} \cdot E[\hat{J}_h(A,B) \mid smallest(i)] \\
&= \textstyle\sum_{i \in A \cap B} \frac{1}{N} \cdot 1 + \sum_{i \in (A \cup B) - (A \cap B)} \frac{1}{N} \cdot 0 \\
&= \textstyle\frac{1}{N} \cdot \sum_{i \in A \cap B} 1 \\
&= \textstyle\frac{1}{N} |A \cap B| \\
&= \textstyle\frac{|A \cap B|}{|A \cup B|} \\
&= J(A,B)
\end{aligned}$$

□

The above lemma states that $\hat{J}_h(A,B)$ is an unbiased estimator for the Jaccard similarity. Note that evaluating the estimator only requires knowledge of $\hat{h}(A)$ and $\hat{h}(B)$: an entire set is squeezed down to just one integer! However, $\hat{J}_h(A,B)$ is not a good estimator since it is a Bernoullian R.V. and thus has a large variance: in the worst case ($J(A,B) = 0.5$), we have $Var[\hat{J}_h(A,B)] = 0.25$ and thus the expected error (standard deviation) of $\hat{J}_h(A,B)$ is $\sqrt{Var[\hat{J}_h(A,B)]} = 0.5$. This means that on expectation (in the worst case) we are off by 50% from the true value of $J(A,B)$. We know how to solve this issue: just take the average of k independent such estimators, for sufficiently large k.

Let $h_i : [1,n] \to [0,M)$, with $i = 1, \ldots, k$, be k independent hash functions that are (i) 1-uniform, and (ii) collision-free on $A \cup B$. We define the MinHash sketch of a set A to be the k-tuple:

Definition 3.9 (MinHash sketch)**.** $h_{min}(A) = (\hat{h}_1(A), \hat{h}_2(A), \ldots, \hat{h}_k(A))$

In other words: the i-th element of $h_{min}(A)$ is the smallest hash $h_i(x)$, for $x \in A$. Note that the MinHash sketch of a set A can be easily computed in $O(k|A|)$ time, provided that h can be evaluated in constant time. Then, we estimate $J(A, B)$ using the following estimator:

Definition 3.10 (Improved MinHash estimator)**.**

$$J^+(A, B) = \frac{1}{k} \sum_{i=1}^{k} \hat{J}_{h_i}(A, B)$$

In other words, we compute the average of $\hat{J}_{h_i}(A, B)$ for $i = 1, \ldots, k$. Note that the improved MinHash estimator can be computed in $O(k)$ time given the MinHash sketches of two sets.

We can immediately apply the double-sided additive Chernoff-Hoeffding bound (Lemma 2.15) and obtain that $P(|J^+(A, B) - J(A, B)| \geq \epsilon) \leq 2e^{-\epsilon^2 k/2}$ for any desired absolute error $0 \leq \epsilon \leq 1$. Fix now any desired failure probability $0 < \delta \leq 1$. By solving $2e^{-\epsilon^2 k/2} = \delta$ we obtain $k = 2\ln(2/\delta)/\epsilon^2$. We can finally state:

Theorem 3.11. *Fix any desired absolute error $0 \leq \epsilon \leq 1$ and failure probability $0 < \delta \leq 1$. By using $k = \frac{2}{\epsilon^2} \ln(2/\delta) \in O\left(\frac{\log(1/\delta)}{\epsilon^2}\right)$ hash functions, the estimator $J^+(A, B)$ exceeds absolute error ϵ with probability at most δ, i.e.*

$$P(|J^+(A, B) - J(A, B)| \geq \epsilon) \leq \delta$$

To summarize, we can squeeze down any subset of $[1, n]$ to a MinHash sketch of $O\left(\frac{\log(1/\delta)}{\epsilon^2} \log n\right)$ bits so that, later, in $O\left(\frac{\log(1/\delta)}{\epsilon^2}\right)$ time we can estimate the Jaccard similarity between any pair of sets (represented with MinHash sketches) with arbitrarily small absolute error ϵ and arbitrarily small failure probability δ.

Note that it is easy to combine the MinHash sketches of two sets A and B so to obtain the MinHash sketch of $A \cup B$ (similarly, to compute the MinHash sketch of $A \cup \{x\}$ given the MinHash sketch of A): $h_{min}(A \cup B) = (\min\{\hat{h}_1(A), \hat{h}_1(B)\}, \ldots, \min\{\hat{h}_k(A), \hat{h}_k(B)\})$.

3.3 Locality-sensitive hashing (LSH)

Locality-sensitive hash functions are used to accelerate the search of similar elements in a data set. Similarity is usually measured in terms of a distance metric, discussed in the next subsection. See Jure Leskovec, Anand Rajaraman, Jeffrey Ullman. *Mining of Massive Datasets*, `www.mmds.org`, Sections 3.4 - 3.8 for more applications.

3.3.1 Distance metrics

A distance metric over a set A is a function $d : A \times A \to \mathbb{R}$ with the following properties:

- Non-negativity: $d(x, y) \geq 0$
- Identity: $d(x, y) = 0$ iff $x = y$
- Simmetry: $d(x, y) = d(y, x)$
- Triangle inequality: $d(x, z) \leq d(x, y) + d(y, z)$

For example, the *Jaccard distance* $d_J(x, y) = 1 - J(x, y)$ defined over sets is indeed a distance metric (exercise: prove it). Some examples of distances among vectors $x, y \in \mathbb{R}^d$ are:

- L_p norm (or Minkowski distance): $L_p(x,y) = \left(\sum_{i=1}^{d} |x_i - y_i|^p\right)^{1/p}$
- L_2 norm (or Euclidean distance): $L_2(x,y) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$
- L_1 norm (or Manhattan distance): $L_1(x,y) = \sum_{i=1}^{d} |x_i - y_i|$
- L_∞ norm: $L_\infty(x,y) = \max\{|x_1 - y_1|, \ldots, |x_d - y_d|\}$
- Cosine distance: $d_{cos}(x,y) = 1 - cos(x,y) = 1 - \frac{x \cdot y}{\|x\| \cdot \|y\|} = \frac{\sum_{i=1}^{d} x_i y_i}{\sqrt{\sum_{i=1}^{d} x_i^2} \cdot \sqrt{\sum_{i=1}^{d} y_i^2}}$

Between strings, we have:

- Hamming distance between two equal-length strings: $H(s_1, s_2)$ is the number of positions $s_1[i] \neq s_2[i]$ in which the two strings differ. On alphabet $\{0,1\}$ it is equal to $L_1(s_1, s_2)$.
- Edit distance between any two strings: $Ed(s_1, s_2)$ is the minimum number of edits (substitutions, single-character inserts/deletes) that have to be applied to s_1 in order to convert it into s_2.

3.3.2 The theory of LSH

Suppose our task is to find all similar pairs of elements (small $d(x,y)$) in a data set $A \subseteq U$ (U is some universe). While a distance-preserving sketch (e.g. for Jaccard distance) speeds up the computation of $d(x,y)$, we still need to compute $|A|^2$ distances in order to find all similar pairs! On big data sets this is clearly not feasible.

A locality-sensitive hash function for some distance metric $d : U \times U \to \mathbb{R}$ is a function $h : U \to [0, M)$ such that similar elements (i.e. $d(x,y)$ is small) are likely to collide: $h(x) = h(y)$. This is useful to drastically reduce the search space with the following algorithm:

1. Scan the data set A and put each element $x \in A$ in bucket $H[h(x)]$ of a hash table H.
2. Compute distances only between pairs inside each bucket $H[i]$.

Classic hash data structures use $O(m)$ space for representing a set of m elements and support insertions and lookups in $O(1)$ expected time (see Section 2.3.4). More advanced data structures[6] support queries in $O(1)$ worst-case time with high probability. In the following, we will therefore assume these bounds for our hash data structures.

LSH works by first defining a distance threshold t. Ideally, we would like the collision probability to be equal to 0 for pairs such that $d(x,y) > t$ and equal to 1 for pairs such that $d(x,y) \leq t$. For example, using a distance $d : U \times U \to [0,1]$ (e.g. Jaccard distance) the ideal LSH function should work as in Figure 1.

In practice, we are happy with a good approximation:

Definition 3.12. *A (d_1, d_2, p_1, p_2)-sensitive family $\mathcal{H}$ of hash functions is such that, for a uniformly-chosen $g \in \mathcal{H}$, we have:*

- *If $d(x,y) \leq d_1$, then $P(g(x) = g(y)) \geq p_1$.*
- *If $d(x,y) \geq d_2$, then $P(g(x) = g(y)) \leq p_2$.*

[6] Dietzfelbinger, Martin, and Friedhelm Meyer auf der Heide. "A new universal class of hash functions and dynamic hashing in real time." International Colloquium on Automata, Languages, and Programming. Springer, Berlin, Heidelberg, 1990.

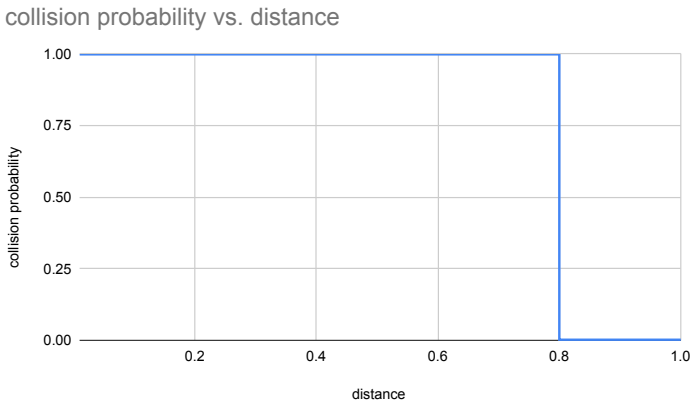

Figure 1: The ideal locality-sensitive hash function: elements whose distance is below the threshold $t = 0.8$ collide with probability 1; elements whose distance is above the threshold do not collide.

Intuitively, we want d_1 and d_2 to be as close as possible ($d_1 \leq d_2$), p_1 as large as possible, and p_2 as small as possible. To abbreviate, in the following we will say that h *is a* (d_1, d_2, p_1, p_2)*-sensitive hash function* when it is uniformly drawn from a (d_1, d_2, p_1, p_2)-sensitive family. For example, Figure 3 shows the behaviour of a $(0.4, 0.7, 0.999, 0.007)$-sensitive hash function for Jaccard distance (see next subsection for more details).

We now show how locality-sensitive hash functions can be *amplified* in order to obtain different (better) parameters.

AND construction Suppose $\mathcal{H}$ is a (d_1, d_2, p_1, p_2)-sensitive family. Pick uniformly r independent hash functions $h_1, \ldots, h_r \in \mathcal{H}$, and define:

Definition 3.13 (AND construction)**.** $h^{AND}(x) = (h_1(x), \ldots, h_r(x))$

Then, if two elements $x, y \in U$ collide with probability p using any of the h_i, now they collide with probability p^k using h^{AND} (because the h_i are independent) and we conclude:

Lemma 3.14. h^{AND} *is a* (d_1, d_2, p_1^k, p_2^k)*-sensitive hash function.*

Observe that, if the output of h is one integer, then h^{AND} outputs r integers. However, we may use one additional collision-free hash function h' to reduce this size to one integer: x is mapped to $y = h'(h^{AND}(x))$. This is important, since later we will need to insert y in a hash table (this trick reduces the space by a factor of r).

OR construction Suppose $\mathcal{H}$ is a (d_1, d_2, p_1, p_2)-sensitive family. Pick uniformly b independent hash functions $h_1, \ldots, h_b \in \mathcal{H}$, and define:

Definition 3.15 (OR construction)**.** *We say that* x *and* y *collide iff* $h_i(x) = h_i(y)$ *for at least one* $1 \leq i \leq b$*.*

Note: the OR construction can be simulated by simply keeping b hash tables $H_1, \ldots, H_b$, and inserting x in bucket $H_i[h_i(x)]$ for each $1 \leq i \leq b$. Then, two elements collide iff they end up in the same bucket in at least one hash table.

Suppose two elements $x, y \in U$ collide with probability p using any hash function h_i. Then:

- For a fixed i, we have that $P(h_i(x) \neq h_i(y)) = 1 - p$

- The probability that all hashes do not collide is $P(\wedge_{i=1}^{b} h_i(x) \neq h_i(y)) = (1-p)^b$
- The probability that at least one hash collides is

$$P(\vee_{i=1}^{b} h_i(x) = h_i(y)) = 1 - P(\wedge_{i=1}^{b} h_i(x) \neq h_i(y)) = 1 - (1-p)^b$$

We conclude:

Lemma 3.16. *The OR construction yields a $(d_1, d_2, 1-(1-p_1)^b, 1-(1-p_2)^b)$-sensitive hash function.*

Combining AND+OR By combining the two constructions, each x is hashed through rb hash functions: we keep b hash tables and insert each $x \in U$ in buckets $H_i[h_i^{AND}(x)]$ for each $1 \leq i \leq b$, where h_i^{AND} is the combination of r independent hash values. We obtain:

Lemma 3.17. *If $\mathcal{H}$ is a (d_1, d_2, p_1, p_2)-sensitive family, then the AND+OR constructions with parameters r and b yields a $(d_1, d_2, 1-(1-p_1^r)^b, 1-(1-p_2^r)^b)$-sensitive family.*

3.3.3 Nearest neighbour search

One application of LSH is nearest neighbour search:

Definition 3.18 (Nearest neighbour search (NNS))**.** *For a given distance threshold D, preprocess a data set A of size $|A| = n$ in a data structure such that later, given any data point x, we can quickly find a point $y \in A$ such that $d(x, y) \leq D$.*

To solve the NNS problem, let $\mathcal{H}$ be a (D', D, p_1, p_2)-sensitive family, with D' as close as possible to (and smaller than) D. Suppose moreover that $h(x)$ can be evaluated in time t_h and $d(x, y)$ can be computed in time t_d (note: t_d can be reduced considerably by employing sketches — see Section 3.2). We amplify $\mathcal{H}$ with an AND+OR construction with parameters r (AND) and b (OR). Our data structure is formed by b hash tables $H_1, \ldots, H_b$. For each of the n data points $x \in A$, we compute the b functions $h_i^{AND}(x)$ in total time $O(n \cdot b \cdot r \cdot t_h)$ and insert in $H_i[h_i^{AND}(x)]$ a pointer to the original data point x (or to its sketch). Assuming that a hash table storing m pointers occupies $O(m)$ words of space and can be constructed in (expected) $O(m)$ time, we obtain:

Lemma 3.19. *Our NNS data structure can be constructed in $O(n \cdot b \cdot r \cdot t_h)$ time and occupies $O(n \cdot b)$ space (in addition to the original data points — or their sketches).*

To answer a query x, note that we are interested in finding just *one* point y such that $d(x, y) \leq D$: we can stop our search as soon as we find one. In $O(t_h \cdot b \cdot r)$ time we compute the hashes $h_i^{AND}(x)$ for all $1 \leq i \leq b$. In the worst case, all the n data points y are such that $d(x, y) > D$. The probability that one such point ends up in bucket $H_i[h_i^{AND}(x)]$ is at most p_2^r. As a result, the expected number of false positive in each bucket $H_i[h_i^{AND}(x)]$ is at most $n \cdot p_2^r$; in total, this yields $n \cdot b \cdot p_2^r$ false positives that need to be checked against x. For each of these false positives, we need to compute a distance in time t_d. We obtain:

Lemma 3.20. *Our NNS data structure answers a query in expected time $O(t_h \cdot b \cdot r + n \cdot b \cdot p_2^r \cdot t_d) = O\big(b(t_h \cdot r + n \cdot p_2^r \cdot t_d)\big)$. If there exists a point within distance at most D' from our query, then we return an answer with probability at least $1-(1-p_1^r)^b$.*

Example 3.21. *Consider the $(0.4, 0.7, 0.999, 0.007)$-sensitive family of Figure 3 (its construction is discussed in the next subsection). This function has been built with AND+OR construction with parameters $r = 10$ and $b = 1200$ starting from a $(0.4, 0.7, 0.6, 0.3)$-sensitive hash function (in fact, $1-(1-0.6^r)^b \approx 0.999$ and $1-(1-0.3^r)^b \approx 0.007$). We can therefore use this hash to solve the NNS problem with threshold $D = 0.7$. Lemma 3.20 states that* at most $100 \cdot 0.3^r \approx 0.0006\%$ *of the n input points are false negatives and need to be checked against our query (compare this with a naive strategy*

that compares 100% *of the points with the query). Moreover, if at least one point within distance* $D' = 0.4$ *from our query exists, we will return a point within distance* 0.7 *with probability at least* $1-(1-0.6^r)^b \approx 0.999$. *The data structure uses space proportional to* $b = 1200$ *words (a few kilobytes) for each data point; note that, in big data scenarios, each data point (for example, a document) is likely to use much more space than that so this extra space is negligible.*

3.3.4 LSH for Jaccard distance

Let $\hat{h}$ be the MinHash function of Definition 3.6. In Section 3.2 we have established that $P(\hat{h}(A) = \hat{h}(B)) = J(A, B)$, i.e. the probability that two elements collide through $\hat{h}$ is exactly their Jaccard similarity. Recall that we have defined the *Jaccard distance* (a metric) to be $d_J(A, B) = 1 - J(A, B)$. But then, $P(\hat{h}(A) = \hat{h}(B)) = 1 - d_J(A, B)$ and we obtain that $\hat{h}$ is a $(d_1, d_2, 1 - d_1, 1 - d_2)$-sensitive hash function for any $0 \leq d_1 \leq d_2 \leq 1$, see Figure 2.

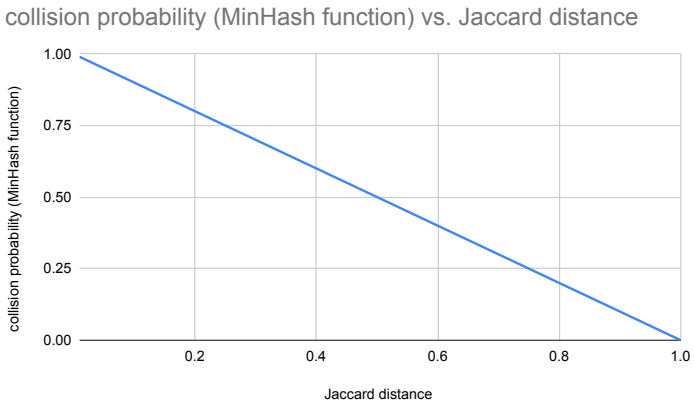

Figure 2: The MinHash function $\hat{h}$ of Definition 3.6 is a $(d_1, d_2, 1 - d_1, 1 - d_2)$-sensitive function for any $0 \leq d_1 \leq d_2 \leq 1$.

Using the AND+OR construction, we can amplify $\hat{h}$ and obtain a $(d_1, d_2, 1 - (1 - (1 - d_1)^r)^b, 1 - (1 - (1 - d_2)^r)^b)$-sensitive function for any $0 \leq d_1 \leq d_2 \leq 1$. For example, with $r = 10$ and $b = 1200$ we obtain a function whose behaviour is depicted in Figure 3.

The shape of the s-curve is dictated by the parameters b and r. As it turns out, b controls the steepness of the slope, that is, the distance between the two points where the probability becomes close to 0 and close to 1. The larger b, the steeper the s-curve is. In other words, b controls the distance between d_1 and d_2 in our LSH: we want b to be large. Parameter r, on the other hand, controls the position of the slope (the point where the curve begins to decrease).

Let p be the collision probability and d_J be the Jaccard distance. The s-curve follows the equation $p = 1-(1-(1-d_J)^r)^b$ By observing that the center of the slope is approximately around $p = 1/2$, one can determine the parameters b and r as a function of the slope position d_J. Let's solve the following equation as a function of r:

$$1 - (1 - (1 - d_J)^r)^b = 1/2$$

We obtain (note that r should be an integer so we must approximate somehow):

$$r = \left\lfloor \frac{\ln\left(1 - 2^{-1/b}\right)}{\ln(1 - d_J)} \right\rfloor$$

The fact that we have to approximate r to an integer means that the slope of the resulting curve will not be centered exactly at d_J. By playing with parameter b, one can further adjust the curve.

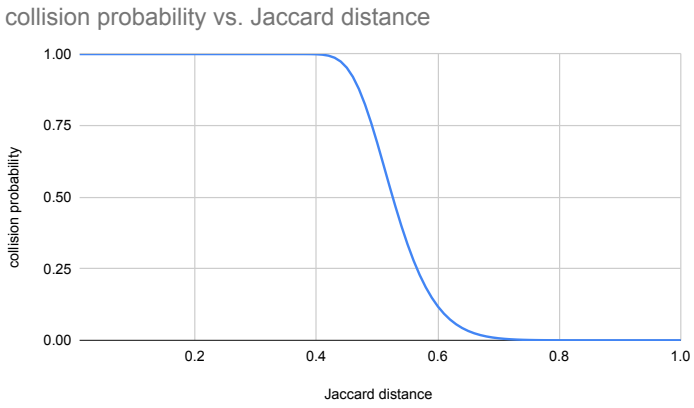

Figure 3: A $(0.4, 0.7, 0.999, 0.007)$-sensitive hash function for Jaccard distance built with AND+OR construction with parameters $r = 10$ and $b = 1200$ starting from a $(0.4, 0.7, 0.6, 0.3)$-sensitive LSH function. Equivalently, we can take two closer points d_1 and d_2 on the curve: for example, this function is also $(0.5, 0.6, 0.69, 0.12)$-sensitive.

Example 3.22. *Suppose we want to build a LSH to identify sets with Jaccard distance at most* 0.9. *We choose a large* $b = 100000$. *Then, the above equation gives us* $r = \left\lfloor \frac{\ln\left(1-2^{-1/100000}\right)}{\ln(1-0.9)} \right\rfloor = 5$. *Using these parameters, we obtain the LSH shown in Figure 4. For example, one can extract two data points from this curve and see that this is a* $(0.85, 0.95, 0.99949, 0.03076)$*-sensitive function.*

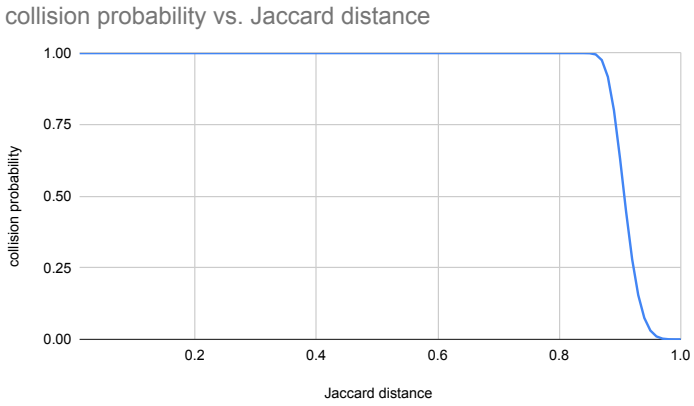

Figure 4: LSH built for Example 3.22.

Clearly, a large b has a cost: in Example 3.22, we have to compute $5 \cdot 10^5$ MinHash functions for each set, which means that we have to apply $5 \cdot 10^5$ basic hash functions h (see Definition 3.6) to each element of each set. These computations however are completely independent, thus they can be parallelized optimally (for example, with a MapReduce job running over a large cluster).

3.3.5 References

Jure Leskovec, Anand Rajaraman, Jeffrey Ullman. *Mining of Massive Datasets*, `www.mmds.org`, chapter 3.

4 Mining data streams

A data stream is a sequence $x = x_1, x_2, \ldots, x_m$ of elements (without loss of generality, integers). We receive these elements one at a time, from x_1 to x_m. Typically, m is too large and we cannot keep all the stream in memory. The goal of streaming algorithms is to compute interesting statistics on the stream while using as little memory as possible (usually, poly-logarithmic in m).

A streaming algorithm is evaluated on these parameters:

1. **Memory** used (as a function of, e.g., m).
2. **Delay** per element: the worst-case time taken by the algorithm to process each stream element.
3. **Probability** of obtaining a correct solution or a good approximation of the correct result.
4. **Approximation ratio** (e.g. the value returned by the algorithm is a $(1 \pm \epsilon)$ approximation of the correct answer, for a small $\epsilon \geq 0$).

4.1 Pattern matching

The first example of stream statistic we consider is *pattern matching*. Say the elements x_i belong to some alphabet Σ: the stream is a string of length m over Σ. Suppose we are given a pattern $y = y_1 y_2 \ldots y_n \in \Sigma^n$. The pattern's length n is smaller than m, but also n could be very large (so that y too does not fit in memory or cache). The question we tackle in this section is: how many times does y appear in x as a substring $y = x_i x_{i+1} \ldots x_{i+n-1}$?

Example 4.1 (Intrusion Detection and Prevention Systems (IDPSs))**.** *IDPSs are software tools that scan network traffic in search of known patterns such as virus fragments or malicious code. The searched patterns are usually very numerous, so the memory usage and delay of the used pattern matching algorithm is critical. Ideally, the algorithm should work entirely in cache in order to achieve the best performance. See also the paper on IDPSs in the references section.*

4.1.1 Karp-Rabin's algorithm

Karp-Rabin hashing is the main tool we will use to solve the problem. First, we note that the technique itself yields a straightforward solution, even though in $O(n)$ space. In the next section we refine this solution to use $O(\log n)$ space.

Suppose we have processed the stream up to $x_1, \ldots, x_i$ $(i \geq n)$ and that we know the hash values $\kappa_{q,z}(x_{i-n+1} x_{i-n+2} \ldots x_i)$ and $\kappa_{q,z}(y)$. By simply comparing these two hash values (in constant time) we can discover whether or not the patter occurs in the last n stream's characters. The crucial step is to update the hash of the stream when a new element x_{i+1} arrives. This is not too hard: we have to subtract character x_{i-n+1} from the stream's hash and add the new character x_{i+1}. This can be achieved as follows:

$$\kappa_{q,z}(x_{i-n+2} x_{i-n+2} \ldots x_{i+1}) = (\kappa_{q,z}(x_{i-n+1} x_{i-n+2} \ldots x_i) - x_{i-n+1} \cdot z^{n-1}) \cdot z + x_{i+1} \mod q$$

The value $z^{n-1} \mod q$ can be pre-computed, so the above operation takes constant time. Note that, since we need to access character x_{i-n+1}, at any time the algorithm must keep the last n characters seen in the stream, thereby using $O(n)$ space.

Analysis Note that, if there are no collisions between the pattern and all the $m - n + 1 \leq m$ stream's substrings of length n, then the algorithm returns the correct result (number of occurrences of the pattern in the stream). From Section 3.1, the probability that the pattern collides with any of those substrings is at most n/q. By union bound, the probability that the patter collides with at least one substring is $mn/q \leq m^2/q$. We want this to happen with small (inverse polynomial probability): this can be achieved by choosing a prime q in the range $[m^{c+2}, 2 \cdot m^{c+2}]$, for any constant c. We obtain:

Theorem 4.2. *The Karp-Rabin algorithm solves the pattern matching problem in the streaming model using $O(n)$ words of memory and $O(1)$ delay. The correct solution is returned with high (inverse-polynomial) probability $1 - m^{-c}$, for any constant $c \geq 1$ chosen at initialization time.*

We note that deterministic algorithms exist for the pattern matching problem in the streaming model, but they have larger delay. For example, it can be proved that Knuth–Morris–Pratt's algorithm has a delay of $O(\log n)$ per stream's character (but overall terminates the task in optimal $O(m)$ time).

4.1.2 Porat-Porat's algorithm

The big disadvantage of Karp-Rabin's algorithm is that it uses too much memory: $O(n)$ words per pattern. In this section we study an algorithm described by Benny Porat and Ely Porat in 2009 that uses just $O(\log n)$ words of space and has $O(\log n)$ delay per stream's character [7]. Other algorithms are able to reduce the delay to the optimal $O(1)$. For simplicity, assume that n is a power of two: $n = 2^e$ for some $e \geq 0$. The algorithm can be generalized to any n in a straightforward way. The overall idea is to:

- Keep a counter *occ* initialized to 0, storing the number of occurrences of y found in x.
- Keep the hashes of all $1 + e = 1 + \log_2 n$ prefixes of y whose length is a power of two.
- Keep the occurrences of those prefixes of y on the stream, working in e levels: level $0 \leq i < e$ records all occurrences of the prefix $y[1, 2^i]$ in a window containing the last 2^{i+1} stream's characters. Using a clever argument based on string periodicity, show that this information can be "compressed" in just $O(1)$ space per level ($O(\log n)$ space in total).
- Before the new character x_j arrives: for every level $i > 0$, if position $p = j - 2^{i+1}$ (the oldest position in the window of level i) was explicitly stored because it is an occurrence of $y[1, 2^i]$, then remove it. In such a case, check also if p is an occurrence of $y[1, 2^{i+1}]$ (do this check using fingerprints). If this is the case, then insert p in level $i + 1$; moreover, if $i + 1 = e$ then we have found an occurrence of y: increment *occ*.
- When the new character x_j arrives: check if it is an occurrence of y_1. If yes, push position j to level 0. Moreover, (cleverly) update the hashes of all positions stored in all levels.

Figure 5 depicts two steps of the algorithm: before and after the arrival of a new stream character. Algorithm 1 implements one step of the above procedure (hiding details such as compression of the occurrences and update of the hashes, which are discussed below). The window at level i is indicated as W_i and it is a set of positions (integers). We assume that the stream is ended by a character # not appearing in the pattern (this is just a technical detail: by the way we define one update step, this is required to perform the checks one last time at the end of the stream).

Compressing the occurrences We have $\log n$ levels, however this is not sufficient to claim that the algorithm uses $O(\log n)$ space: in each level i, there could be up to 2^i occurrences of the pattern's prefix $y[1, 2^i]$. However, in this paragraph we show that all the occurrences in a window can be compressed in just $O(1)$ space.

The key observation is that, in each level, we store occurrences of the pattern's prefix of length $K = 2^i$ in a window of size $2K = 2^{i+1}$. Now, if there are at least three such occurrences, then at least two of them must overlap. But these are occurrences of the same string $y[1, 2^i]$, so if they overlap the string must be periodic. Finally, if the string is periodic then all its occurrences in the window must be equally-spaced: we have an occurrence every p positions, for some integer p (a period of the string). Then, all occurrences in the window can be stored in just $O(1)$ space: just remember the

[7] Note that, no matter how large n is, $O(\log n)$ words will fit in cache. $O(\log n)$ delay in cache is by far more desirable than $O(1)$ delay in RAM: the former is hundreds of times faster than the latter.

Algorithm 1: new_stream_character(x_j)

```
foreach level i = 0, ..., e − 1 do
    if j − 2^{i+1} ∈ W_i then
        W_i ← W_i − {j − 2^{i+1}};
        if κ_{q,z}(y[1, 2^{i+1}]) == κ_{q,z}(x[j − 2^{i+1}, j − 1]) then
            if i == e − 1 then
                occ ← occ + 1;
            else
                W_{i+1} ← W_{i+1} ∪ {j − 2^{i+1}};
if x_j == y_1 then
    W_0 ← W_0 ∪ {j};
```

first occurrence r_1, the period p, and the number t of occurrences. This representation is also easy to update (in constant time) upon insertion of new occurrences to the right (which must follow the same rule) and removal of an occurrence to the left. We now formalize this reasoning.

Definition 4.3 (Period of a string)**.** *Let S be a string of length K. We say that S has period p if and only if $S[i] = S[i+p]$ for all $1 \leq p \leq K - p$.*

Example 4.4. *The string $S = abcabcabcabca$, of length $K = 13$, has periods 3, 6, 9, 12.*

Theorem 4.5 (Wilf's theorem)**.** *Any string having periods p, q and length at least $p + q - gcd(p, q)$ also has $gcd(p, q)$ as a period.*

Example 4.6. *Consider the string above: $S = abcabcabcabca$. The string has periods 6, 9 (with $gcd(6, 9) = 3$) and has length $13 > 6 + 9 - 3 = 12$. Wilf's theorem can be used to deduce that the string must also have period $gcd(6, 9) = 3$.*

Wilf's theorem can be used to prove the following (exercise):

Lemma 4.7. *Let S be a string of length K, and P be a string of length $2K$. If S occurs in P at positions $r_1 < r_2 < \cdots < r_q$, with $q \geq 3$, then $r_{j+1} = r_j + p$, where $p = r_2 - r_1$.*

The lemma provides a compressed representation for all the occurrences in a window: just record (r_1, p, q). This representation is easy to update in constant time whenever an occurrence exits/enters the window.

Updating the fingerprints The only thing left to show is how to efficiently compute $\kappa_{q,z}(x[j - 2^{i+1}, j - 1])$ at level i, that is, the fingerprint of the whole window, when the first occurrence stands at the beginning of the window: $r_1 = j - 2^{i+1}$. Consider the window W_i at level i, and the two smallest positions $r_1, r_2 \in W_i$. Let $x = x[1, j-1]$ be the current stream. We keep in memory three fingerprints:

(i) $\kappa_{q,z}(x)$: the fingerprint of the whole stream.

(ii) $\kappa_{q,z}(x[r_1, r_2 - 1])$: the stream's substring standing between r_1 (included) and r_2 (excluded).

(iii) $\kappa_{q,z}(x[1, r_1 - 1])$: the fingerprint of the stream's prefix beginning before r_1.

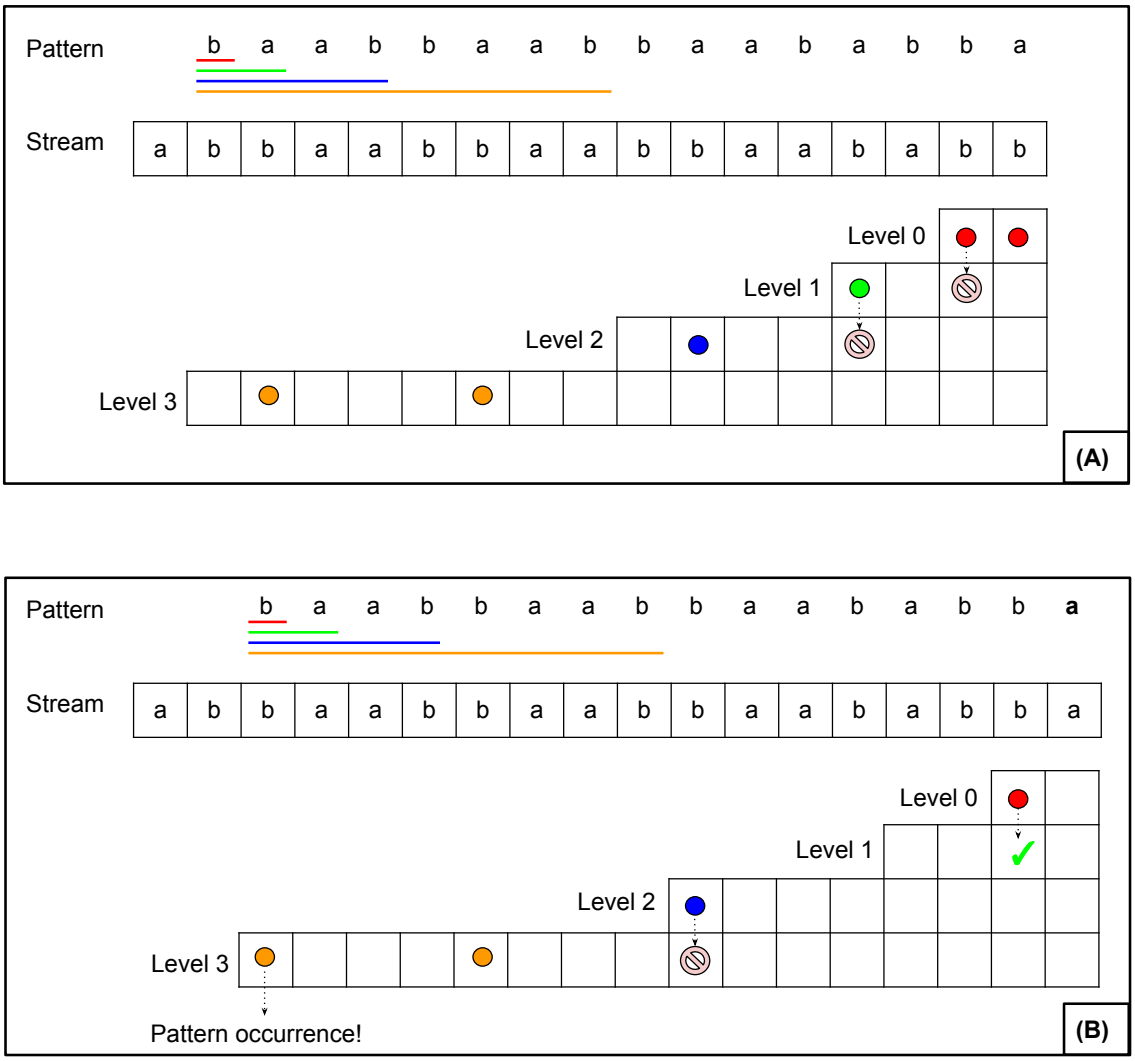

Figure 5: Each colored dot at level i represents an occurrence of a prefix of length 2^i of the pattern (underlined with corresponding color). **(A)** We have two such occurrences at level 0, one occurrence at levels 1 and 2, and two occurrences at level 3. Some of these occurrences are candidates that could be promoted to the next level: these are the leftmost occurrences at levels 0 and 1. Unfortunately, none of these occurrences can be promoted since they are not occurrences of a prefix of length 2^{i+1} of the pattern: the leftmost occurrence at level 0 is not an occurrence of "ba" and the occurrence at level 1 is not an occurrence of "baab". They will therefore be eliminated before the next stream's character arrives. **(B)** A new stream character ("a", in bold) has arrived. All occurrences, except the eliminated ones, have been shifted to the left in the levels' windows. Now, three occurrences are candidates that could be promoted to the next level. The occurrence at level 0 is indeed an occurrence of "ba", therefore it will be promoted to level 1 before the next stream's character arrives. The occurrence at level 2 is not an occurrence of "baabbaab", therefore it will be eliminated. Finally, the leftmost occurrence at level 3 is an occurrence of the full pattern: before removing it from the window, we will increment the counter *occ*.

(i) can be clearly updated in constant time each time a new stream character arrives (see Section

3.1).

Fingerprint (ii) can be computed when the second occurrence enters the window as $y[1, 2^i] = W_i[j - 2^i, j - 1]$ (i.e. $r_2 = j - 2^i$). Then one can verify that:

$$\kappa_{q,z}(x[r_1, r_2 - 1]) = \left(\kappa_{q,z}(x) - \kappa_{q,z}(y[1, 2^i]) - \kappa_{q,z}(1, r_1 - 1) \cdot z^{2^i + (r_2 - r_1)}\right) \cdot z^{-2^i} \mod q$$

In the above equation, the quantity $z^{2^i+(r_2-r_1)} \mod q$ can be computed one step at a time (i.e. multiplying z by itself $2^i + (r_2 - r_1)$ times modulo q) while the stream characters between r_1 and $r_2 + 2^i - 1$ arrive (constant time per character per level). The $\log n$ values $z^{-2^i} \mod q$ can be precomputed before the stream arrives in $O(\log m \log n)$ time as follows. $z^{2^{i+1}} \equiv_q (z^{2^i})^2$, and $z^{-2^i} \mod q$ can be computed in $O(\log q) = O(\log m)$ time from $z^{-2^i} \mod q$ using Euler's theorem and fast exponentiation: $a^{-1} \equiv_q a^{q-2}$.

Fingerprint (iii): when the first occurrence enters the window as $W_i[j - 2^i, j - 1]$ (i.e. $r_1 = j - 2^i$), this fingerprint is $\kappa_{q,z}(x[1, r_1 - 1]) = (\kappa_{q,z}(x) - \kappa_{q,z}(y[1, 2^i])) \cdot z^{-2^i}$. We only need to show how to update this fingerprint when r_1 leaves the window and is replaced by r_2. This is easy to achieve using (ii):

$$\kappa_{q,z}(x[1, r_2 - 1]) = \kappa_{q,z}(x[1, r_1 - 1]) \cdot z^{r_2 - r_1} + \kappa_{q,z}(x[r_1, r_2 - 1]) \mod q$$

where $z^{r_2-r_1} \mod q$ is computed as $z^{r_2-r_1} \equiv_q z^{2^i+(r_2-r_1)} \cdot z^{-2^i}$ (both computed at step (ii)).

We are finally ready to show how to compute the fingerprint of the whole window when r_1 stands at its beginning:

$$\kappa_{q,z}(x[r_1, j - 1]) = \kappa_{q,z}(x[j - 2^{i+1}, j - 1]) = \kappa_{q,z}(x) - \kappa_{q,z}(x[1, r_1 - 1]) \cdot z^{2^{i+1}} \mod q$$

We obtain:

Theorem 4.8. *The Porat-Porat algorithm solves the pattern matching problem in the streaming model using $O(\log n)$ words of memory and $O(\log n)$ delay. The correct solution is returned with high (inverse-polynomial) probability $1 - m^{-c}$, for any constant $c \geq 1$ chosen at initialization time.*

4.1.3 References

V. Gupta, M. Singh and V. K. Bhalla, "Pattern matching algorithms for intrusion detection and prevention system: A comparative analysis," 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2014, pp. 50-54.

B. Porat and E. Porat. Exact and approximate pattern matching in the streaming model. In FOCS 2009, pages 315–323. IEEE, 2009.

4.2 Counting with small registers: Morris' algorithm

In 1978, Robert Morris was working at Bell labs and faced the problem of counting large numbers using very small (8 bits) registers. Using just 8 bits, the largest number that can be stored is clearly 255. However, as Morris noticed, this is true only if we wish to store *exact* counts; if we allow for some error, then the register can actually hold larger numbers.

In general, in order to count up to n we need $\log n$ bits (logarithms are base 2). What if we allow for a 2-approximation, that is, we allow our register to be off by at most a factor of two? then, it is easy to see that $\log \log n$ bits are sufficient: instead of storing our number $x < n$, we store the largest power of two not exceeding it: $2^y \leq x$, with $y = \lfloor \log x \rfloor$. Since y takes integer values between 0 and $\log n$, it only takes at most $\log \log n$ bits to store it. In general, we may fix a relative error $0 < \epsilon \leq 1$ and approximate x with the largest power of $(1 + \epsilon)$ not exceeding it: $(1 + \epsilon)^y \leq x$, with

$y = \lfloor \log_{1+\epsilon} x \rfloor = \left\lfloor \frac{\log x}{\log(1+\epsilon)} \right\rfloor$. Then, y requires just $\log n - \log\log(1+\epsilon) = \log n + O(\log\log(1/\epsilon))$ bits to be stored (we used the bound $\log(1+\epsilon) \geq \frac{1}{1/\epsilon+1/2}$ and assumed $0 < \epsilon < 1$) and is a ϵ-approximation of x.

Example 4.9. *Suppose we allow for a* 10% *relative error (i.e.* $\epsilon = 0.1$*). Then, we need approximately just* $\log\log n + 3$ *bits. What is the largest number we can store in 8 bits? Solving* $\log\log n + 3 = 8$ *we obtain* $\log\log n = 5$*, i.e. we can store a number as large as* $2^{2^5} = 2^{32}$ *with 10% relative error.*

While the above reasoning shows how one can store approximately a large counter in a small number of bits, it does not show how to *increment* such counter: in real-case applications (such as the ones considered by Morris in his original article), we may wish to start from an approximate counter initialized with 0 and increase it one unit at a time (for example, every time a certain event occurs). Morris' idea is to employ *randomization*, incrementing the counter with small probability. By keeping enough parallel counters and averaging out the results, we can get the desired approximation with high enough probability.

4.2.1 The basic algorithm

Here's the basic algorithm using one register. In the next sections we will initialize several parallel versions of the algorithm to further boost the success probability.

Algorithm 2: `Morris`

input : A stream of n events and a desired relative error $0 < \epsilon \leq 1$
output: A $(1 \pm \epsilon)$-approximation $\hat{n}$ of the number of events in the stream, with failure probability $1/(2\epsilon^2)$

1 Initialize a register $X = 0$;
2 For each event to be counted: increment X with probability 2^{-X};
3 Finally, output $2^X - 1$;

4.2.2 Analysis - unbiased estimator

The first thing to prove is that our estimator $2^X - 1$ is unbiased:

Lemma 4.10. $E[2^X - 1] = n$

Proof. We proceed by induction on n. Let us denote with X_i the register's content after event i. For $n = 0$, we have $X_0 = 0$ and $E[2^{X_0} - 1] = 0$ so we are done. Assume inductively that the claim holds for n, i.e. $E[2^{X_n} - 1] = n$ (equivalently, $E[2^{X_n}] = n + 1$). Then:

$$\begin{aligned}
E[2^{X_{n+1}} - 1] &= E[2^{X_{n+1}}] - 1 \\
&= \textstyle\sum_{i=0}^{\infty} P(X_n = i) \cdot E[2^{X_{n+1}} \mid X_n = i] - 1 \\
&= \textstyle\sum_{i=0}^{\infty} P(X_n = i) \cdot (2^{-i} \cdot 2^{i+1} + (1 - 2^{-i}) \cdot 2^i) - 1 \\
&= \textstyle\sum_{i=0}^{\infty} P(X_n = i) \cdot (2^i + 1) - 1 \\
&= \textstyle\sum_{i=0}^{\infty} P(X_n = i) \cdot 2^i + \sum_{i=0}^{\infty} P(X_n = i) - 1 \\
&= \textstyle\sum_{i=0}^{\infty} P(X_n = i) \cdot 2^i \\
&= E[2^{X_n}] = n + 1
\end{aligned}$$

□

where in the second step we applied the law of total expectation (Lemma 2.9).

4.2.3 Analysis of the basic algorithm

Having established that the expected value of our estimator is exactly the count n that we wish to store, we only miss to establish how much a single realization of the estimator can differ from the expected value. We first compute the estimator's variance:

Lemma 4.11. $Var[2^X - 1] \leq n^2/2$

Proof.

$$\begin{aligned} Var[2^X - 1] &= E[((2^X - 1) - n)^2] \\ &= E[(2^X - (n+1))^2] \\ &= E[2^{2X}] - 2(n+1)E[2^X] + (n+1)^2 \\ &= E[2^{2X}] - (n+1)^2 \end{aligned} \tag{13}$$

Let X_n denote the value of our register after seeing n events. We now compute $E[2^{2X_n}]$ and plug it into Equation 13.

$$\begin{aligned} E[2^{2X_n}] &= \textstyle\sum_{i=0}^{\infty} P(X_n = i) \cdot 2^{2i} \\ &= \textstyle\sum_{i=0}^{\infty} 2^{2i} \cdot \big(P(X_{n-1} = i-1) \cdot 2^{-(i-1)} + P(X_{n-1} = i) \cdot (1 - 2^{-i})\big) \\ &= \textstyle\sum_{i=0}^{\infty} 2^{i+1} \cdot P(X_{n-1} = i-1) + \sum_{i=0}^{\infty} 2^{2i} \cdot P(X_{n-1} = i) - \sum_{i=0}^{\infty} 2^{i} \cdot P(X_{n-1} = i) \\ &= \textstyle\sum_{i=0}^{\infty} 4 \cdot 2^{i-1} \cdot P(X_{n-1} = i-1) + \sum_{i=0}^{\infty} 2^{2i} \cdot P(X_{n-1} = i) - \sum_{i=0}^{\infty} 2^{i} \cdot P(X_{n-1} = i) \\ &= 4 \cdot E[2^{X_{n-1}}] + E[2^{2X_{n-1}}] - E[2^{X_{n-1}}] \\ &= 3 \cdot E[2^{X_{n-1}}] + E[2^{2X_{n-1}}] \\ &= 3n + E[2^{2X_{n-1}}] \end{aligned}$$

The above yields a recursive definition: denoting $E_n = E[2^{2X_n}]$, we have $E_n = 3n + E_{n-1}$. Since $E_0 = E[2^{2 \cdot 0}] = 1$, this series expands to $E_n = \sum_{i=1}^{n} 3i + 1 = \frac{3n(n+1)}{2} + 1$. We can finally plug this into Equation 13 and obtain

$$\begin{aligned} Var[2^X - 1] &= E[2^{2X}] - (n+1)^2 \\ &= \tfrac{3n(n+1)}{2} + 1 - (n+1)^2 \\ &= (n^2 - n)/2 \leq n^2/2 \end{aligned}$$

□

Our bound on the variance can be now plugged into a Chebyshev bound in order to get our first (weak) bound on the relative error. For any $0 < \epsilon < 1$:

$$P(|(2^X - 1) - n| \geq n \cdot \epsilon) \leq \frac{Var[2^X - 1]}{n^2 \epsilon^2} \leq \frac{1}{2\epsilon^2}$$

Note that the above bound is not very informative: for $\epsilon < 1/\sqrt{2}$, the probability on the right-hand side is greater than 1. We are now going to boost the success probability with two standard tricks: average of independent instances of the algorithm, followed by median of several averages. Our final goal will be to obtain an algorithm achieving the following guarantee: for any choice of ϵ (relative error) and δ (failure probability), the algorithm will satisfy:

$$P(\text{relative error } > \epsilon) \leq \delta$$

This is a standard goal in probabilistic approximation algorithms. Clearly, this improvement will not come for free: we will need to run $f(\epsilon, \delta)$ independent parallel instances of Morris' algorithm, each with its own independent register, for some function f of the two parameters ϵ and δ.

4.2.4 A first improvement: Morris+

A first improvement, indicated here with the name *Morris+*, can be achieved by simply applying Lemma 2.14 to the results of $s = \frac{3}{2\epsilon^2} = O(1/\epsilon^2)$ independent (parallel) instances of the algorithm. Let us denote with Θ^+ the mean of the s results. Then, Lemma 2.14 (boosted Chebyshev) yields:

$$P(|\Theta^+ - n| > n \cdot \epsilon) \leq \frac{1}{s \cdot 2\epsilon^2} \leq \frac{1}{3}$$

This bound is clearly better than the previous one: we fail with constant probability 1/3, no matter the desired relative error. Note that the price to pay is a higher space usage: now we need to keep $s = O(1/\epsilon^2)$ independent registers. Note also that the error could be made arbitrarily small by increasing s: the space increases linearly with s, and the failure probability decreases linearly with s. In the next paragraph we show how to achieve an exponentially-decreasing failure probability with one final (standard) trick: taking the median of independent instances of Morris+.

4.2.5 Final algorithm: Morris++

The final step is to execute t independent parallel instances of Morris+. Note that each of the t instances consists of s parallel instances of (the basic) Morris' algorithm, so in the end we will have st parallel instances (each with its own independent register). Let us denote with $\Theta_1^+, \ldots, \Theta_t^+$ the t instances of Morris+. The output of our algorithm Morris++ is the *median* $\Theta^{++} = \text{median}\{\Theta_1^+, \ldots, \Theta_t^+\}$ of the t instances. We now show that the median is indeed correct (has relative error bounded by ϵ) with high probability, and we compute the total space usage (i.e. number of register) required in order to guarantee relative error ϵ with probability at least $1 - \delta$, for any choice of ϵ and δ. We state the result as a general Lemma, since it will be useful also in other results.

Lemma 4.12. *Let Θ^+ be an estimator such that $E[\Theta^+] = n$ and $P(|\Theta^+ - n| > n \cdot \epsilon) \leq \frac{1}{3}$. For any desired failure probability δ, draw $t = 72\ln(1/\delta)$ instances of the estimator and define $\Theta^{++} = \text{median}\{\Theta_1^+, \ldots, \Theta_t^+\}$. Then:*

$$P(|\Theta^{++} - n| > n \cdot \epsilon) \leq \delta$$

Proof. Consider the following indicator random variable:

$$\mathbb{1}_i = \begin{cases} 1 & \text{if } |\Theta_i^+ - n| \geq n \cdot \epsilon \\ 0 & \text{otherwise} \end{cases}$$

That is, $\mathbb{1}_i$ is equal to 1 if and only if Θ_i^+ fails, i.e. if its relative error exceeds ϵ. From the previous subsection, note that $\mathbb{1}_i$ takes value 1 with probability at most 1/3.

What is the probability that Θ^{++} fails, i.e. that $|\Theta^{++} - n| \geq \epsilon \cdot n$? If the median fails, then it is either too small (below $(1 - \epsilon) \cdot n$) or too large (above $(1 + \epsilon) \cdot n$). In either case, by definition of median, at least $t/2$ estimators Θ_i^+ return a result which is too small or too large, and thus fail. In other words,

$$P(|\Theta^{++} - n| \geq \epsilon \cdot n) \leq P\left(\sum_{i=1}^{t} \mathbb{1}_i \geq t/2\right)$$

As seen above, each $\mathbb{1}_i$ is a Bernoullian R.V. taking value 1 with probability at most 1/3. We thus have $\mu = E[\sum_{i=1}^{t} \mathbb{1}_i] \leq t/3$. Clearly, decreasing μ decreases also the probability that $\sum_{i=1}^{t} \mathbb{1}_i$ exceeds $t/2$ (see also the following Chernoff-Hoeffding bound), so for simplicity in the following calculations we will consider $\mu = t/3$ (we aim at an upper-bound to this probability so this simplification is safe).

Recall the one-sided right variant of the Chernoff-Hoeffding additive bound (Lemma 2.15):

$$P\left(\sum_{i=1}^{t} \mathbb{1}_i \geq \mu + k\right) = P\left(\sum_{i=1}^{t} \mathbb{1}_i \geq \frac{t}{3} + k\right) \leq e^{\frac{-k^2}{2t}}$$

Solving $t/3 + k = t/2$, we obtain $k = t/6$. Replacing this value into the previous inequality, we obtain:

$$P\left(\sum_{i=1}^{t} \mathbb{1}_i \geq t/2\right) \leq e^{-t/72}$$

We want the probability on the right-hand side to equal our parameter δ. Solving $\delta = e^{-t/72}$ we obtain $t = 72\ln(1/\delta)$. □

Since we previously established that Morris+ uses $s = 3/(2\epsilon^2)$ registers, in total Morris++ uses $st = O(\epsilon^{-2}\ln(1/\delta))$ registers. The last thing to notice is that each register stores a number (X_n) whose expected value is $\log n$, so each register requires on expectation $\log\log n$ bits. We can state our final result:

Theorem 4.13. *For any desired relative error $0 < \epsilon \leq 1$ and failure probability $0 < \delta < 1$, the algorithm Morris++ uses $O\left(\frac{\log(1/\delta)}{\epsilon^2}\log\log n\right)$ bits on expectation and, with probability at least $1-\delta$, counts numbers up to n with relative error at most ϵ, i.e. it returns a value Θ^{++} such that:*

$$P(|\Theta^{++} - n| > n \cdot \epsilon) \leq \delta$$

4.2.6 References

Morris, R. (1978). Counting large numbers of events in small registers. Communications of the ACM, 21(10), 840–842.

`http://gregorygundersen.com/blog/2019/11/11/morris-algorithm/`

Jelani Nelson's Lecture 1 of CS229r: Algorithms for Big Data `http://people.seas.harvard.edu/~minilek/cs229r/fall15/lec.html`

4.3 Counting distinct elements

In this section we consider the following problem. Suppose we observe a stream of m integers $x_1, x_2, x_3, \ldots, x_m$ (arriving one at a time) from the interval $x_i \in [1, n]$ and we want to count the number of *distinct* integers in the stream, i.e. $d = |\{x_1, x_2, \ldots, x_m\}|$. We cannot afford to use too much memory (and m and d are very large — typically in the order of billions).

Here are reported some illuminating examples of the practical relevance of the count-distinct problem. Some of these examples are taken from the paper from Estan et. al that can be found in the references section.

Example 4.14 (DoS attacks)**.** *Denial of Service attacks can be detected by analyzing the number of distinct flows (source-destination IP pairs contained in the headers of TCP/IP packets) passing through a network hub in a specific time interval. The reason is that typical DoS software use large numbers of fake IP sources; if they were to use few IP sources, then those sources could be easily identified (and blocked) because of the large traffic they must generate in order for the DoS attack to be effective.*

Example 4.15 (Spreading rate of a worm)**.** *Worms are self-replicating malware whose goal is to spread to as many computers as possible using a network (e.g. the Internet) as medium. In order to count how many computers have been infected by the worm, one needs to (1) filter packets containing the worm's code, and (2) count the number of distinct source IPs in the headers of those packets. From `https://www.caida.org/archive/code-red/` (an analysis of the spread of the Code-Red version 2 worm between midnight UTC July 19, 2001 and midnight UTC July 20, 2001):*

"On July 19, 2001 more than 359,000 computers were infected with the Code-Red (CRv2) worm in less than 14 hours. At the peak of the infection frenzy, more than 2,000 new hosts were infected each minute."

Example 4.16 (Distinct IPs)**.** *Suppose we wish to count how many people are visiting our web site (or sending requests to our server). Then, we need to count how many* distinct *IP numbers are connecting to the server that hosts the web site. For example, this problem becomes non-trivial (large m and d) for search engines.*

Example 4.17 (String complexity)**.** *A random string (for simplicity, binary) of length m has the property that the number of distinct substrings $T(k)$ of length $k \geq \log_2 m$ is approximately $m - k$. The more repetitive the string is, the smaller this number becomes. As it turns out, $T(k)/k$ is a good indicator of the complexity (or compressibility) of the string. This is again a count-distinct problem in the stream model: compute $T(k)/k$ for a streamed string.*

4.3.1 Naive solutions

A first naive solution to the count-distinct problem is to keep a bitvector $B[1, n]$ of n bits, initialized with all 0's. Then it is sufficient to set $B[x_i] = 1$ for each element x_i of the stream. Finally, we count the number of 1's in the bitvector. If n is very large (like in typical applications), this solution uses too much space. A second solution could be to store the stream elements in a self-balancing binary search tree or in a hash table with dynamic re-allocation. This solution uses $O(d \log n)$ bits of space, which could still be too much if the number d of distinct elements is very large. In the next sections we will see how to compute an approximate answer within *logarithmic* space. We first discuss Flajolet-Martin's idealized algorithm, which however assumes a uniform hash function and thus cannot actually be implemented in small space. Then, we study a practical variant (bottom-k) which only requires two-independent hash functions and can thus be implemented in truly logarithmic space.

4.3.2 Flajolet-Martin's algorithm

The following solution is due to Philippe Flajolet and G. Nigel Martin (1983). Let $[1, n]$ denote the range of integers $1, 2, \ldots, n$ and $[0, 1]$ denote the range of all real values between 0 and 1, included. In this section we study an idealized version that uses a uniform hash function $h : [1, n] \to [0, 1]$ ("idealized" because such a function would actually require $\Theta(n)$ words of space to be stored, see Section 2.3).

Algorithm 3: `FM`

input : A stream of integers $x_1, \ldots, x_m$.
output: An estimate $\hat{d}$ of the number of distinct integers in the stream.

1 Initialize $y = 1$;
2 For each stream element x, update $y \leftarrow \min(y, h(x))$;
3 When the stream ends, return the estimate $\hat{d} = \frac{1}{y} - 1$;

Intuitively, why does FM work? First, note that repeated occurrences of some integer x in the stream will yield the same hash value $h(x)$. Since h is uniform, we end up drawing d uniform real numbers $y_1 < y_2 < \cdots < y_d$ in the interval $[0, 1]$ [8]. At the end, the algorithm returns $1/y_1 - 1$. The more distinct y_i's we see, the more likely it is to see a smaller value. In particular, h will spread the y_i's uniformly in the interval $[0, 1]$; think, for a moment, about the most "uniform" (regular) way to spread those numbers in $[0, 1]$: this happens when the intervals $[0, y_1]$, $[y_i, y_{i+1}]$, $[y_d, 1]$ have all the

[8] Note that we can safely assume $x \neq y \Rightarrow h(x) \neq h(y)$: since we draw uniform numbers on the real line, the probability that $h(x) = h(y)$ is zero.

same length $y_{i+1} - y_i = y_1 - 0 = y_1 = 1/(d+1)$. But then, our claim $1/y_1 - 1 = d$ follows. It turns out that this is true also *on average* (not just in this idealized "regular" case): the *average* distance between 0 and the smallest hash y_1 seen in the stream is precisely $1/(d+1)$. Next, we prove this intuition.

Lemma 4.18. *Let $y = \min\{h(x_1), \ldots, h(x_m)\}$. Then, $E[y] = 1/(d+1)$.*

Proof.

$$\begin{aligned} E[y] &= \int_0^1 P(y \geq \lambda)\, d\lambda && \text{Lemma 2.6} \\ &= \int_0^1 P(\forall x_i \ : \ h(x_i) \geq \lambda)\, d\lambda \\ &= \int_0^1 (1-\lambda)^d\, d\lambda && h \text{ is uniform} \\ &= -\tfrac{(1-\lambda)^{d+1}}{d+1} \Big|_0^1 \\ &= \tfrac{1}{d+1} \end{aligned}$$

□

Unfortunately, in general $E[1/y] \neq 1/E[y]$ so it is not true that $E[\hat{d}] = E[1/y - 1] = d$. Technically, we say that $\hat{d}$ is not an unbiased estimator for d. On the other hand, if y is very close to $1/(d+1)$, then intuitively also $\hat{d}$ will be very close to d. We will prove this intuition by studying the relative error of y with respect to $1/(d+1)$, and then turn this into a relative error of $\hat{d}$ with respect to d.

Lemma 4.19. *Let $y = \min\{h(x_1), \ldots, h(x_m)\}$. Then, $Var[y] \leq 1/(d+1)^2$.*

Proof. We use the equality $Var[y] = E[y^2] - E[y]^2$. We know that $E[y]^2 = 1/(d+1)^2$. We compute $E[y^2]$ as follows:

$$\begin{aligned} E[y^2] &= \int_0^1 P(y^2 \geq \lambda)\, d\lambda \\ &= \int_0^1 P(y \geq \sqrt{\lambda})\, d\lambda \\ &= \int_0^1 (1-\sqrt{\lambda})^d\, d\lambda \end{aligned}$$

We can solve the latter integral by the substitution $u = 1 - \sqrt{\lambda}$. We have $\lambda = (1-u)^2$ and $\frac{d\lambda}{du} = d(1-u)^2/du = -2(1-u)$, so $d\lambda = -2(1-u)\, du$. Also, note that $u = 0$ for $\lambda = 1$ and $u = 1$ for $\lambda = 0$ so the integral's interval switches. By applying the substitution we obtain:

$$\begin{aligned} E[y^2] &= \int_0^1 (1-\sqrt{\lambda})^d\, d\lambda \\ &= \int_1^0 -2(1-u)u^d\, du \\ &= -2\left(\int_1^0 u^d\, du - \int_1^0 u^{d+1}\, du\right) \\ &= -2\left(\tfrac{u^{d+1}}{d+1}\Big|_1^0 - \tfrac{u^{d+2}}{d+2}\Big|_1^0\right) \\ &= -2\left(-\tfrac{1}{d+1} + \tfrac{1}{d+2}\right) \\ &= \tfrac{2}{d+1} - \tfrac{2}{d+2} \end{aligned}$$

To conclude:

$$\begin{aligned} Var[y] &= E[y^2] - E[y]^2 \\ &= \tfrac{2}{d+1} - \tfrac{2}{d+2} - \tfrac{1}{(d+1)^2} \\ &= \tfrac{2(d+2)-2(d+1)}{(d+1)(d+2)} - \tfrac{1}{(d+1)^2} \\ &= \tfrac{2}{(d+1)(d+2)} - \tfrac{1}{(d+1)^2} \\ &\leq \tfrac{2}{(d+1)^2} - \tfrac{1}{(d+1)^2} \\ &= \tfrac{1}{(d+1)^2} \end{aligned}$$

□

From here, we proceed as in Section 4.2. We define an algorithm `FM+` that computes the mean $y' = \sum_{i=1}^{s} y_i/s$ of s independent parallel instances of `FM`, for some s to be determined later. Applying Chebyshev (Lemma 2.14), we obtain

$$P\left(\left|y' - \frac{1}{d+1}\right| > \frac{\epsilon}{d+1}\right) \leq \frac{1}{(d+1)^2} \cdot \frac{(d+1)^2}{s\epsilon^2} = \frac{1}{s\epsilon^2}$$

Our algorithm `FM+` returns $\hat{d}' = 1/y' - 1$. How much does this value differ from the true value d? Note that the above inequality gives us $\frac{1-\epsilon}{d+1} \leq y' \leq \frac{1+\epsilon}{d+1}$ with probability at least $1 - \frac{1}{s\epsilon^2}$. Let us assume $0 < \epsilon < 1/2$. In this range, the following inequality holds: $\frac{1}{1-\epsilon} \leq 1 + 2\epsilon$. We have:

$$\begin{aligned} \tfrac{1}{y'} - 1 &\leq \tfrac{d+1}{1-\epsilon} - 1 \\ &\leq (1+2\epsilon)(d+1) - 1 \\ &= d + 2\epsilon d + 2\epsilon \\ &\leq d + 4\epsilon d \\ &= d(1+4\epsilon) \end{aligned}$$

Similarly, in the interval $0 < \epsilon < 1/2$ the following inequality holds: $\frac{1}{1+\epsilon} \geq 1 - \epsilon$. We have:

$$\begin{aligned} \tfrac{1}{y'} - 1 &\geq \tfrac{d+1}{1+\epsilon} - 1 \\ &\geq (1-\epsilon)(d+1) - 1 \\ &\geq d(1-2\epsilon) \end{aligned}$$

Thus, `FM+` returns a $(1 \pm 4\epsilon)$ approximation with probability at least $1 - 1/(s\epsilon^2)$ for any $0 < \epsilon < 1/2$. To obtain a $(1 \pm \epsilon)$-approximation, we simply adjust ϵ (i.e. turn to a relative error $\epsilon' = 4\epsilon$) and obtain that `FM+` returns a $(1 \pm \epsilon)$-approximation with probability at least $1 - 16/(s\epsilon^2)$ for any $0 < \epsilon < 1$. Finally, we apply the median trick (Lemma 4.12). We first force the failure probability to be 1/3:

$$\frac{16}{s\epsilon^2} = \frac{1}{3} \Leftrightarrow s = \frac{48}{\epsilon^2}$$

Our final algorithm `FM++` runs $t = 72\ln(1/\delta)$ parallel instances of `FM+` and returns the median result. From Lemma 4.12 we obtain:

Theorem 4.20. *For any desired relative error $0 < \epsilon \leq 1$ and failure probability $0 < \delta < 1$, with probability at least $1 - \delta$ the `FM++` algorithm counts the number d of distinct elements in the stream with relative error at most ϵ, i.e. it returns a value $\bar{d}$ such that:*

$$P(|\bar{d} - d| > \epsilon \cdot d) \leq \delta$$

In order to achieve this result, during its execution `FM++` needs to keep in memory $O\left(\frac{\ln(1/\delta)}{\epsilon^2}\right)$ hash values.

4.3.3 Bottom-k algorithm

Motivated by the fact that a uniform $h : [1, n] \to [0, 1]$ takes too much space to be stored (see Section 2.3), in this section we present an algorithm that only requires a two-independent hash function $h : [1, n] \to [0, 1]$. See Section 2.3.3 for a discussion on how to implement such a function in practice.

The Bottom-k algorithm is presented as Algorithm 4. It is a generalization of Flajolet-Martin's algorithm: we keep the smallest k distinct hash values $y_1 < y_2 < \cdots < y_k$ seen in the stream so far, and finally return the estimate k/y_k. In our analysis we will show that, by choosing $k \in O(\epsilon^{-2})$, we obtain a ϵ-approximation with constant probability. Finally, we will boost the success probability with a classic median trick.

Algorithm 4: `Bottom-k`

input : A stream of integers $x_1, \ldots, x_m$ and a desired relative error $\epsilon \leq 1/2$.

output: A $(1 \pm \epsilon)$-approximation $\hat{d}$ of the number of distinct integers in the stream, with failure probability 1/3.

1 Choose $k = 24/\epsilon^2$;
2 Initialize $(y_1, y_2, \ldots, y_k) = (1, 1, \ldots, 1)$;
3 For each stream element x, update the k-tuple $(y_1, y_2, \ldots, y_k)$ with the new hash $y = h(x)$ so that the k-tuple stores the k smallest hashes seen so far;
4 When the stream ends, return the estimate $\hat{d} = k/y_k$;

Analysis Crucially, note that the proof of the following lemma will only require two-independence of our basic discrete hash function h'.

Lemma 4.21. *For any $\epsilon \leq 1/2$, Algorithm 4 outputs an estimator $\hat{d}$ such that*

$$P(|\hat{d} - d| > \epsilon \cdot d) \leq 1/3$$

Proof. We first compute one side of the inequality: $P(\hat{d} > (1 + \epsilon)d)$. Let $z_1, \ldots, z_d$ be the d distinct integers in the stream, sorted arbitrarily. Let X_i be an indicator 0/1 variable defined as $X_i = 1$ if and only if $h(z_i) < \frac{k}{d(1+\epsilon)}$. Observe that, if $\sum_{i=1}^{d} X_i \geq k$, then at the end of the stream the smallest k hash values must satisfy $y_1 < y_2 < \cdots < y_k < \frac{k}{d(1+\epsilon)}$. But then, the returned estimate is $\hat{d} = k/y_k > d(1+\epsilon)$. The converse is also true: if $\hat{d} = k/y_k > d(1 + \epsilon)$, then $y_k < \frac{k}{d(1+\epsilon)}$, thus $y_1 < y_2 < \cdots < y_k < \frac{k}{d(1+\epsilon)}$ and then $\sum_{i=1}^{d} X_i \geq k$. To summarize:

$$\textstyle\sum_{i=1}^{d} X_i \geq k \text{ if and only if } \hat{d} > d(1 + \epsilon)$$

We can therefore reduce our problem to an analysis of the random variable $\sum_{i=1}^{d} X_i$. Since $h(z_i)$ is uniform in $[0, 1]$, $P\left(h(z_i) < \frac{k}{d(1+\epsilon)}\right) = \frac{k}{d(1+\epsilon)} = p$. X_i is a Bernoullian R.V. with success probability p, so $E[X_i] = p = \frac{k}{d(1+\epsilon)}$. By linearity of expectation:

$$E\left[\sum_{i=1}^{d} X_i\right] = \frac{k}{1 + \epsilon}$$

The variance of this R.V. is also easy to calculate. Note that, since the $h(z_i)$'s are pairwise independent, then the X_i's are pairwise independent (in addition to being identically distributed) and we can apply

Lemma 2.8 to $Var\left[\sum_{i=1}^{d} X_i\right]$. Recall also (Corollary after Lemma 2.11) that $Var[X_i] \leq E[X_i]$. We obtain:

$$Var\left[\sum_{i=1}^{d} X_i\right] = \sum_{i=1}^{d} Var[X_i] \leq \sum_{i=1}^{d} E[X_i] = E\left[\sum_{i=1}^{d} X_i\right] = \frac{k}{1+\epsilon} \leq k$$

We can now apply Chebyshev to $\sum_{i=1}^{d} X_i$:

$$P\left(\left|\sum_{i=1}^{d} X_i - \frac{k}{1+\epsilon}\right| > \sqrt{6k}\right) \leq \frac{Var\left[\sum_{i=1}^{d} X_i\right]}{(\sqrt{6k})^2} \leq \frac{k}{6k} = 1/6$$

In particular, we can remove the absolute value:

$$P\left(\sum_{i=1}^{d} X_i - \frac{k}{1+\epsilon} > \sqrt{6k}\right) \leq 1/6 \Leftrightarrow P\left(\sum_{i=1}^{d} X_i > \sqrt{6k} + \frac{k}{1+\epsilon}\right) \leq 1/6$$

For which k does it hold that $\sqrt{6k} + \frac{k}{1+\epsilon} \leq k$? a few manipulations give

$$k \geq \frac{6(1+\epsilon)^2}{\epsilon^2}$$

Moreover: $\frac{6(1+\epsilon)^2}{\epsilon^2} \leq \frac{6(1+1)^2}{\epsilon^2} = \frac{24}{\epsilon^2}$. Therefore, if we choose $k = 24/\epsilon^2$ then $\sqrt{6k} + \frac{k}{1+\epsilon} \leq k$ and:

$$P\left(\sum_{i=1}^{d} X_i > k\right) \leq P\left(\sum_{i=1}^{d} X_i > \sqrt{6k} + \frac{k}{1+\epsilon}\right) \leq 1/6$$

We finally obtain $P(\hat{d} > (1+\epsilon)d) \leq 1/6$.

We are now going to prove the symmetric inequality $P(\hat{d} < (1-\epsilon)d) \leq 1/6$. The proof will proceed similarly to the previous case. Let $z_1, \ldots, z_d$ be the d distinct integers in the stream, sorted arbitrarily. Let X_i be an indicator 0/1 variable defined as $X_i = 1$ if and only if $h(z_i) > \frac{k}{d(1-\epsilon)}$. Observe that, if $\sum_{i=1}^{d} X_i > d-k$, then at the end of the stream the largest $(d-k)+1$ hash values must be larger than $\frac{k}{d(1-\epsilon)}$. In particular, the k-th smallest hash y_k is also larger than this value: $y_k > \frac{k}{d(1-\epsilon)}$. But then, the returned estimate is $\hat{d} = k/y_k < d(1-\epsilon)$. The converse is also true: if $\hat{d} = k/y_k < d(1-\epsilon)$, then $y_k > \frac{k}{d(1-\epsilon)}$. Since y_k is the k-th smallest hash value, all the following (larger) $d-k$ hash values must also be larger than $\frac{k}{d(1-\epsilon)}$, i.e. $\sum_{i=1}^{d} X_i > d-k$. To summarize:

$$\sum_{i=1}^{d} X_i > d-k \text{ if and only if } \hat{d} < d(1-\epsilon)$$

Note that $X_i \sim Be\left(1 - \frac{k}{d(1-\epsilon)}\right)$, so $E[X_i] = 1 - \frac{k}{d(1-\epsilon)}$. The expected value of $\sum_{i=1}^{d} X_i$ is:

$$E\left[\sum_{i=1}^{d} X_i\right] = dE[X_i] = d - \frac{k}{1-\epsilon}$$

Recall (Corollary after Lemma 2.11) that $Var[X_i] \leq 1 - E[X_i]$. Recalling that we assume $\epsilon \leq 1/2$, we have:

$$Var\left[\sum_{i=1}^{d} X_i\right] = dVar[X_i] \leq d(1 - E[X_i]) = d \cdot \frac{k}{d(1-\epsilon)} \leq 2k$$

By Chebyshev:

$$P\left(\left|\sum_{i=1}^{d} X_i - \left(d - \frac{k}{1-\epsilon}\right)\right| > \sqrt{12k}\right) \leq \frac{2k}{12k} = 1/6$$

Removing the absolute value and re-arranging terms:

$$P\left(\sum_{i=1}^{d} X_i > \sqrt{12k} + d - \frac{k}{1-\epsilon}\right) \leq 1/6$$

For which values of k do we have $\sqrt{12k} + d - \frac{k}{1-\epsilon} \leq d - k$? after a few manipulations, we get

$$k \geq \frac{12(1-\epsilon)^2}{\epsilon^2}$$

Moreover, $\frac{12(1-\epsilon)^2}{\epsilon^2} \leq 12/\epsilon^2$. Therefore, choosing $k = 24/\epsilon^2 > 12/\epsilon^2$, we have $\sqrt{12k} + d - \frac{k}{1-\epsilon} \leq d - k$. Then:

$$P\left(\sum_{i=1}^{d} X_i > d - k\right) \leq P\left(\sum_{i=1}^{d} X_i > \sqrt{12k} + d - \frac{k}{1-\epsilon}\right) \leq 1/6$$

We conclude that $P(\hat{d} < (1-\epsilon)d) \leq 1/6$. Combining this with $P(\hat{d} > (1+\epsilon)d) \leq 1/6$ by union bound, we finally obtain the two-sided bound $P(|\hat{d} - d| > \epsilon \cdot d) \leq 1/3$. □

At this point, we are in the same exact situation of Section 4.2.5: we have an algorithm that achieves relative error ϵ with probability at least 2/3. We can therefore apply the median trick (Lemma 4.12): we run $t = 72\ln(1/\delta)$ parallel instances of our algorithm, and return the median result. Let us call `Bottom-k+` the resulting algorithm. Recall that one hash value takes $O(\log n)$ bits to be stored, and that we keep in total $kt \in O(\log(1/\delta)/\epsilon^2)$ hash values (t instances of `Bottom-k`, keeping k hash values each). Lemma 4.12 allows us to conclude:

Theorem 4.22. *For any desired relative error $0 < \epsilon \leq 1/2$ and failure probability $0 < \delta < 1$, the `Bottom-k+` algorithm uses $O\left(\frac{\log(1/\delta)}{\epsilon^2} \log n\right)$ bits and, with probability at least $1 - \delta$, counts the number d of distinct elements in the stream with relative error at most ϵ, i.e. it returns a value $\bar{d}$ such that:*

$$P(|\bar{d} - d| > \epsilon \cdot d) \leq \delta$$

Example 4.23. *Let's assume we wish to estimate how many distinct IPv4 addresses (32 bits each) are visiting our website. Then, $n = 2^{32}$. Say we choose a function $h' : [1, n] \to [0, M]$ that is collision-free with probability at least $1 - n^2$. Then (see beginning of this section), $M = n^4$ and each hash value requires $\log_2 M = 4\log_2 n = 128$ bits (16 bytes) to be stored. We want `Bottom-k+` to return an answer that is within 10% of the correct answer ($\epsilon = 0.1$, $1/\epsilon^2 = 100$) with probability at least $1 - 10^{-5}$ ($\delta = 10^{-5}$, $\ln(1/\delta) < 12$). Then, replacing the constants that pop up from our analysis we obtain that `Bottom-k+` uses at most around 32 MiB of RAM.*

Note that to prove our main Theorem 4.22 we used rather loose upper bounds. Still, `Bottom-k+`'s memory usage of < 32 MiB is rather limited if compared with the naive solutions. A bitvector of length n would require 4 GiB of RAM. On the other hand, C++'s `std::set` uses 32 bytes per distinct element[9], so it is competitive with our analysis of `Bottom-k+` only for d up to $\approx 10 \cdot 10^5$; this is clearly not sufficient in big-data scenarios such as a search engine: with over 5 billion searches per day[10], Google would need gigabytes of RAM to solve the problem with a `std::set` (even assuming as many as 10 searches per distinct user, and even using more space-efficient data structures). Even better, practical optimized implementations of distinct-count algorithms solve the same problem within few kilobytes of memory[11].

[9] `https://lemire.me/blog/2016/09/15/the-memory-usage-of-stl-containers-can-be-surprising/`
[10] `https://review42.com/resources/google-statistics-and-facts`
[11] `https://en.wikipedia.org/wiki/HyperLogLog`

4.3.4 References

Estan, Cristian, George Varghese, and Mike Fisk. "Bitmap algorithms for counting active flows on high speed links." Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement. 2003.

Alex Andoni's Lecture 4 of COMS 4995-3: Advanced Algorithms (2017) `http://www.cs.columbia.edu/~andoni/s17_advanced/algorithms/mainSpace/Lectures%20%26%20Resources.html`

P. Flajolet and G. N. Martin. Probabilistic counting. In Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS), pages 76-82, 1983.

Seth Gilbert's Lecture 4 of CS 5234 - Algorithms at Scale (2019) `https://www.comp.nus.edu.sg/~gilbert/CS5234/`

Jelani Nelson's lecture of COMPSCI 229r (2016) `https://www.youtube.com/watch?v=fvZ_6pvP4jc&t=1s`