Randomized Algorithms

Karl Stratos

1 Set Membership

Let $S = \{x_1 \ldots x_n\} \subset \mathcal{X}$ where n is very large, and consider the task of checking $y \in S$. A naive approach is to store S in a list and compare y against every $x \in S$, taking $O(n)$ space and time. A better approach is to store S in a hash table, taking $O(1)$ time on average through hashing (Appendix A). But the space complexity remains $O(n)$ since a hash table must keep all original elements to resolve collisions.

If we ignore collisions, we only need $O(m)$ space for a fixed hash table size m. This simplified data structure is called a **Bloom filter** [1]. More specifically,

1. We do not keep the actual elements in S. To check $y \in S$, we use only a binary vector $b \in \{0,1\}^m$ where $b_i = 1$ indicates that *some* $x \in S$ has been binned into the i-th bucket. Thus we may get false positives.

2. There is no false negative.

3. The query time is $O(1)$ in the worst case.

We can reduce the chance of false positives by using k independent hash functions $h_1 \ldots h_k : \mathcal{X} \to \{1 \ldots m\}$. An example with $S = \{a, b, c\}$ may look like ($m = 8$, $k = 2$)

	$b = (0,0,0,0,0,0,0,0)$
Insert(a): $h_1(a) = 2$ and $h_2(a) = 6$	$b = (0,1,0,0,0,1,0,0)$
Insert(b): $h_1(b) = 6$ and $h_2(b) = 3$	$b = (0,1,1,0,0,1,0,0)$
Insert(c): $h_1(c) = 1$ and $h_2(c) = 7$	$b = (1,1,1,0,0,1,1,0)$
IsMember(e): $h_1(e) = 4$ and $h_2(e) = 6 \Rightarrow$ `False` (true negative)	$b = (1,1,1,0,0,1,1,0)$
IsMember(f): $h_1(f) = 1$ and $h_2(f) = 3 \Rightarrow$ `True` (false positive)	$b = (1,1,1,0,0,1,1,0)$

We see that $e \notin S$ avoided being a false positive even though one hash collided ($b_{h_2(e)} = 1$). In general, $y \notin S$ becomes a false positive iff all k hashes collide. This gives the standard false positive rate analysis:

$$\begin{aligned}
\Pr\left(\text{“}y \in S\text{”}|k\right) &= \Pr\left(b_{h_j(y)} = 1 \text{ for all } j \in \{1 \ldots k\}\right) \\
&= \prod_{j=1}^{k} \Pr\left(b_{h_j(y)} = 1\right) && \text{(since } h_1(y) \ldots h_k(y) \in \{1 \ldots m\} \text{ are independent)} \\
&= \prod_{j=1}^{k} \left(\frac{1}{m} \sum_{i=1}^{m} \Pr\left(b_i = 1\right)\right) && \text{(since } h_j \text{ is uniformly random)} \\
&= \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k && \text{(since } \Pr\left(b_i = 0\right) = (1 - 1/m)^{nk} \text{ for any } i) \\
&\approx \left(1 - e^{-\frac{n}{m}k}\right)^k && \text{(since } e^{-z} \approx 1 - z \text{ for } z \approx 0) \qquad (1)
\end{aligned}$$

We can minimize (1) over $k > 0$ (ignoring wholeness) to approximate the optimal number of hash functions as a function of “bits per element” $\frac{m}{n}$. Even though (1) is nonconvex, it is minimized to $2^{-(\ln 2)\frac{m}{n}}$ at $k^\star = (\ln 2)\frac{m}{n}$ (Lemma B.1). For instance, we can achieve the false positive rate of 1% using 9.6 bits per element assuming an optimal number of independent hash functions ($k \approx 7$).

2 Set Similarity

Given sets $S, S' \subset \{1 \dots m\}$, a standard measurement of their similarity is **Jaccard similarity**,

$$\mathrm{JS}(S, S') := \frac{|S \cap S'|}{|S \cup S'|} \in [0, 1] \tag{2}$$

which is 0 iff S and S' are disjoint and 1 iff $S = S'$. Computing (2) takes $O(m)$ space and time. Our goal is to reduce this complexity to $O(1)$ (at query time) by sampling.

2.1 MinHash

If we view (2) as a probability, it is amenable to Monte Carlo methods. For instance, we can sample uniformly at random from $S \cup S'$ for k times and return the fraction of samples in $S \cap S'$ as an unbiased estimate of (2). However, doing this naively gives no computational advantage over exact calculation. **MinHash** [2] is a clever technique that allows for efficient estimation, though it requires a pre-query computation (signature matrix construction) that takes $O(m)$ time. Deriving MinHash involves multiple steps.

2.1.1 Problem reformulation

First, we view a set S as a binary feature vector $\phi(S) \in \{0, 1\}^m$ where $\phi_i(S) = 1$ iff $i \in S$. Let $\Phi = [\phi(S), \phi(S')] \in \{0, 1\}^{m \times 2}$. There are three types of rows in Φ:

- $\mathcal{A} \subset \{1 \dots m\}$: the set of a rows equal to $[1, 1]$
- $\mathcal{B} \subset \{1 \dots m\}$: the set of b rows equal to $[1, 0]$ or $[0, 1]$
- $\mathcal{C} \subset \{1 \dots m\}$: the set of c rows equal to $[0, 0]$

Since they are disjoint, we have $a + b + c = m$. Now the task of estimating (2) can be framed as sampling from $\mathcal{A} \cup \mathcal{B}$ and computing the fraction of rows landing in $\mathcal{A}$, since this converges to $\frac{a}{a+b} = \frac{|S \cap S'|}{|S \cup S'|} = \mathrm{JS}(S, S')$.

2.1.2 Random permutation

Next, we use a random permutation π of $\{1 \dots m\}$ to simulate the sampling process. Let $\Phi^\pi = [\phi^\pi(S), \phi^\pi(S')]$ denote the row-permuted matrix where $\Phi^\pi_{\pi(i)} = \Phi_i$. We define a "signature" $c_\pi(S) \in \{1 \dots m\}$ of the set S based on the permuted feature vector $\phi^\pi(S)$ as follows:

$$c_\pi(S) = \min\left\{j : \phi^\pi_j(S) = 1\right\} = \min\left\{\pi(i) : \ \phi_i(S) = 1\right\} \tag{3}$$

(i.e., the earliest position of 1 in $\phi^\pi(S)$). We may view the signature matrix $C^\pi = [c_\pi(S), c_\pi(S')] \in \{1 \dots m\}^{1 \times 2}$ as a random 1-dimensional projection of $\Phi \in \{0, 1\}^{m \times 2}$. An example with $m = 6$, $S = \{4\}$, and $S' = \{3, 4, 6\}$ is

$$\Phi = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \qquad \pi = \begin{bmatrix} 4 \\ 1 \\ 6 \\ 5 \\ 3 \\ 2 \end{bmatrix} \qquad \Phi^\pi = \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{1} \\ 0 & 0 \\ 0 & 0 \\ \mathbf{1} & 1 \\ 0 & 1 \end{bmatrix} \qquad C^\pi = \begin{bmatrix} \mathbf{5} & \mathbf{2} \end{bmatrix}$$

Theorem 2.1 (The MinHash Theorem)**.** With randomness over the permutation π of $\{1 \dots m\}$,

$$\Pr\left(c_\pi(S) = c_\pi(S')\right) = \mathrm{JS}(S, S')$$

Proof. Let $i_\pi = \min\left\{c_\pi(S), c_\pi(S')\right\}$ (e.g., 2 for the green row in Φ^π above). Then

$$\Pr\left(c_\pi(S) = c_\pi(S')\right) = \Pr\left(\Phi^\pi_{i_\pi} = [1, 1]\right) = \frac{a}{a + b}$$

The last equality holds because (i) $\Phi^\pi_{i_\pi} \notin \mathcal{C}$ by definition (3), and (ii) π is uniformly random over $\mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$. This implies that i_π is uniformly random over $\mathcal{A} \cup \mathcal{B}$. □

Remark 2.2. It is crucial that the same permutation π is used to permute $\phi(S)$ and $\phi(S')$ together. In contrast, there is nothing special about taking the *earliest* position of 1 in the permuted feature vectors. A "MaxHash" taking the *latest* position of 1 would be equally valid.

The estimator. Assuming k independent random permuations $\Pi = (\pi_1 \dots \pi_k) \in \{1 \dots m\}^{k \times m}$, let $C^\Pi \in \{1 \dots m\}^{k \times 2}$ denote the signature matrix whose r-th row is $[c_{\pi_r}(S), c_{\pi_r}(S')]$. The multi-sample estimator is

$$\widehat{\mathrm{JS}}_k(S, S') = \frac{1}{k} \sum_{r=1}^{k} [[c_{\pi_r}(S) = c_{\pi_r}(S')]] \tag{4}$$

(i.e., the agreement rate between the two columns of C^Π). Since $\widehat{\mathrm{JS}}(S, S') \in [0, 1]$ is bounded and has the mean $\mathrm{JS}(S, S')$ by Theorem 2.1, a trivial application of Hoeffding's inequality gives us for all $\epsilon, \delta > 0$,

$$k > \frac{1}{2\epsilon^2} \ln \frac{2}{\delta} \qquad \Rightarrow \qquad \Pr\left(\left| \widehat{\mathrm{JS}}_k(S, S') - \mathrm{JS}(S, S') \right| < \epsilon \right) > 1 - \delta$$

(e.g., setting $\epsilon = 0.1$ and $\delta = 0.01$, we have that using $k = 265$ random permuations is sufficient to guarantee with 99% confidence that the estimate is off by at most 0.1).

2.1.3 Constructing the signature matrix

More generally, we have $n \geq 2$ nonempty sets $S_1 \dots S_n \subset \{1 \dots m\}$ and wish to estimate Jaccard similarity for any pair. Using k random permutations Π, once we have the corresponding signature matrix $C^\Pi \in \{1 \dots m\}^{k \times n}$, estimating Jaccard similarity for any pair at query time requires only $O(k)$ time and space by (4). Since $c_\pi(S) = \min\{\pi(i) : \phi_i(S) = 1\}$ (3), we can construct C^Π without storing $\Phi^\pi \in \{0, 1\}^{m \times n}$ (the intermediate permuted matrix) for each $\pi \in \Pi$ in a streaming fashion as follows:

Input: $\Phi = [\phi(S_1) \dots \phi(S_n)] \in \{0, 1\}^{m \times n}$, $k \ll m$
Output: $C^\Pi \in \{1 \dots m\}^{k \times n}$

- Compute k random permutations $\Pi = (\pi_1 \dots \pi_k) \in \{1 \dots m\}^{k \times m}$.
- Initialize $C^\Pi \leftarrow \infty^{k \times n}$.
- For $i = 1 \dots m$:
  - For $j = 1 \dots n$ such that $\phi_i(S_j) = 1$:

$$C^\Pi_{r,j} \leftarrow \min(C^\Pi_{r,j}, \pi_r(i)) \qquad \forall r = 1 \dots k$$

The construction takes $O(mnk)$ time and $O(mk + nk)$ space. We can reduce the memory overhead by using a linear hash function $h_r : \mathbb{Z} \to \{1 \dots m\}$ to simulate a permutation π_r (Appendix A.2). This yields the final hash-based version:

Input: $\Phi = [\phi(S_1) \dots \phi(S_n)] \in \{0, 1\}^{m \times n}$, $k \ll m$
Output: $C^H \in \{1 \dots m\}^{k \times n}$

- Initialize k independent random linear hash functions $h_1 \dots h_k : \mathbb{Z} \to \{1 \dots m\}$ parameterized by coefficients $H \in \mathbb{R}^{k \times 2}$ simulating k random permutations.
- Initialize $C^H \leftarrow \infty^{k \times n}$.
- For $i = 1 \dots m$:
  - Precompute the hash values $h_1(i) \dots h_k(i) \in \{1 \dots m\}$.
  - For $j = 1 \dots n$ such that $\phi_i(S_j) = 1$:

$$C^H_{r,j} \leftarrow \min(C^H_{r,j}, h_r(i)) \qquad \forall r = 1 \dots k$$

While it has the same asymptotic complexity (for $m < n$), it completely avoids the overhead of computing and storing $\Pi \in \{1 \dots m\}^{k \times m}$.

2.2 MinHash LSH

A common use of MinHash is quickly finding similar documents in a large pool of n documents. In this setting, a document $S \subset \{1 \dots m\}$ is viewed as a bag of m words, featurized as $\phi(S) \in \{0, 1\}^m$. For some suitable number of hash functions k, we can construct the signature matrix $C^H \in \{1 \dots m\}^{k \times n}$ in $O(mnk)$ time and $O(nk)$ space using MinHash. (When a new document comes in, we can add it to the matrix in $O(mk)$ time and $O(k)$ space.)

Assuming we have $C^H \in \{1 \dots m\}^{k \times n}$, we are interested in finding all document pairs S_i, S_j such that $\mathrm{JS}(S_i, S_j) > \tau$ for some threshold $\tau \in [0, 1]$. A naive way of achieving this is to compute $\widehat{\mathrm{JS}}_k(S_i, S_j) = \frac{1}{k}\sum_{r=1}^{k}[[C^H_{r,i} = C^H_{r,j}]]$ for all i, j which takes $O(kn^2)$ time, and collecting i, j such that $\widehat{\mathrm{JS}}_k(S_i, S_j) > \tau$ as an approximation. This is intractable for a large value of n. Instead, **locality-sensitive hashing (LSH)** is used to approximate these pairs in $O(kn)$ time. LSH refers to the broad family of fuzzy hashing techniques that hash similar input items into the same buckets with high probability.

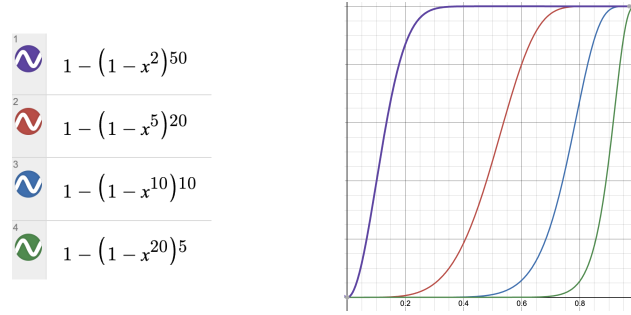
Specifically, we hash $C^H \in \{1 \dots m\}^{k \times n}$ *again*. Divide the rows into B "bands" of width M (thus $k = BM$). Choose B independent vector-valued hash functions $g_1 \dots g_B : \{1 \dots m\}^M \to \{1 \dots P\}$ (e.g., (10)) where $P = O(n)$ is the number of LSH buckets. Hash the bands in C^H to obtain a new matrix $D \in \{1 \dots P\}^{B \times n}$, taking $O(kn)$ time. Finally, take any i, j such that $D_{b,i} = D_{b,j}$ for *some* $b \in \{1 \dots B\}$ as a candidate pair. Then we have

$$\Pr(i \text{ and } j \text{ become a candidate pair}) = 1 - \left(1 - \mathrm{JS}(S_i, S_j)^M\right)^B$$

where the randomness is over the random permutations in MinHash.[1] Thus given a pair of documents with Jaccard similarity $s \in [0, 1]$, we can compute the probability of them becoming a candidate pair under LSH by

$$f(s) = 1 - \left(1 - s^M\right)^B$$

It turns out that f forms an S-shaped curve on $[0, 1]$ for any choice of M and B. Below we plot it for some choices for $k = 100 = MB$:



It is easy (though tedious) to show that $f''(s^\star) = 0$ (i.e., f changes the fastest) at

$$s^\star = \left(\frac{M-1}{MB-1}\right)^{1/M} \approx \left(\frac{1}{B}\right)^{1/M}$$

where the approximation holds for large M. The idea is that two documents are likely to become a candidate pair if their Jaccard similarity is at least $(1/B)^{1/M}$. This gives a principled strategy to select B and M to optimize the chance of collecting pairs with similarity $> \tau$. Specifically, given the target threshold τ, choose B and M (satisfying $k = BM$) such that $(1/B)^{1/M} \approx \tau$.

3 Sampling Without Replacement

Sampling $k \leq n$ elements from an array $x = (x_1 \dots x_n)$ without replacement can be framed as sampling a random subset $R \subseteq \{1 \dots n\}$ with $|R| = k$. Let $\mathcal{R}$ denote all $\binom{n}{k}$ possible outcomes of R. We want to define a distribution p_R of $R \in \mathcal{R}$ such that it contains any of the n positions with equal probability.[2] Formally,

$$\Pr(i \in R) = \sum_{r \in \mathcal{R}:\, i \in r} p_R(r) = \frac{k}{n} \qquad \forall i \in \{1 \dots n\} \tag{5}$$

[1] A standard way to see this is to note that (1) $s_{i,j} := \mathrm{JS}(S_i, S_j)$ is the probability that i and j agree in any row of $C^H \in \{1 \dots m\}^{k \times n}$, (2) $s_{i,j}^M$ is the probability that i and j agree in an entire band, (3) $1 - s_{i,j}^M$ is the probability that i and j disagree in a band, (4) $(1 - s_{i,j}^M)^B$ is the probability that i and j disagree in all B bands, and thus (5) $1 - (1 - s_{i,j}^M)^B$ is the probability that i and j agree in some band (i.e., i and j become a candidate pair).

[2] We can check that this probability must be $\frac{k}{n}$. We have n equalities $\Pr(i \in R) = \sum_{r: i \in r} p_R(r) = p$ for some p. Since any $r \in \mathcal{R}$ contains k distinct positions, $p_R(r)$ is used k times in these equations. Thus their sum is $k(\sum_r p_R(r)) = np$, or $p = \frac{k}{n}$.

A strictly stronger condition is $R \sim \text{unif}(\mathcal{R})$.[3] Then

$$\Pr(i \in R) = \sum_{r \in \mathcal{R}:\, i \in r} \frac{1}{\binom{n}{k}} = \frac{\binom{n-1}{k-1}}{\binom{n}{k}} = \frac{k}{n}$$

However, explicitly considering $\binom{n}{k}$ members of $\mathcal{R}$ takes $O((\frac{n}{k})^k)$ time and space. We can instead directly sample without replacement in $O(k)$ time and $O(n)$ space as follows. Store an array $z = (1 \ldots n)$ of the n positions. For $j = 1 \ldots k$: (1) sample $I_j \sim \text{Unif}(\{j \ldots n\})$, (2) swap z_j and z_{I_j}. For $n = 5$ and $k = 3$, this my look like

$$\begin{array}{lcr} j = 1: \; I_1 = 4 \sim \text{Unif}(\{1,2,3,4,5\}) & \Rightarrow & z = (4,2,3,1,5) \\ j = 2: \; I_2 = 2 \sim \text{Unif}(\{2,3,4,5\}) & \Rightarrow & z = (4,2,3,1,5) \\ j = 3: \; I_3 = 4 \sim \text{Unif}(\{3,4,5\}) & \Rightarrow & z = (4,2,1,3,5) \end{array}$$

Let Z denote the resulting random z. For $j \leq k$, the only way $Z_j = i$ is if i is not selected for the first $j-1$ rounds and then selected in the j-th round. In the above example, $Z_3 = 1$ because 1 is not selected in round 1 (with probability $\frac{4}{5}$) and round 2 (with probability $\frac{3}{4}$), then selected in round 3 (with probability $\frac{1}{3}$). The probability of the event is thus $\frac{1}{5}$. Note that the swapping mechanism is crucial for these probabilities to hold. More generally,

$$\Pr(Z_j = i) = \left(\frac{n-1}{n}\right) \times \left(\frac{n-2}{n-1}\right) \times \cdots \times \left(\frac{n-j+1}{n-j+2}\right) \times \left(\frac{1}{n-j+1}\right) = \frac{1}{n} \qquad \forall j \in \{1 \ldots k\},\; i \in \{1 \ldots n\} \tag{6}$$

Taking $R = \{Z_1 \ldots Z_k\}$ satisfies (5) since

$$\Pr(i \in R) = \Pr(Z_j = i \text{ for some } j \in \{1 \ldots k\}) \stackrel{*}{=} \sum_{j=1}^{k} \Pr(Z_j = i) = \frac{k}{n}$$

where $\stackrel{*}{=}$ holds because $Z_j = i$ and $Z_t = i$ are disjoint events for $j \neq t$. On the other hand, having a random subset R satisfying (5) does not imply a random array Z satisfying (6), so the algorithm solves a strictly harder problem.

In-place shuffling Our intention is to sample k elements from $x = (x_1 \ldots x_n)$, so we can directly work with x in place, thereby avoiding the $O(n)$ space overhead needed for storing $z = (1 \ldots n)$. The algorithm is

Shuffle(x, k)

- For $j = 1 \ldots k$:

$$I_j \sim \text{Unif}(\{j \ldots N\})$$
$$x_j, x_{I_j} \leftarrow x_{I_j}, x_j$$

Shuffle(x, k) reorders x so that $x_1 \ldots x_k$ correspond to any of the n positions in the original array. In particular, **Shuffle**(x, n) randomly permutes x.

3.1 Reservoir Sampling

We now wish to sample k elements from a streaming array $x_1, x_2, \ldots$ without replacement. For $t = 1, 2, \ldots$, we want any of $x_1 \ldots x_t$ to be included with equal probability. Again, this can be framed as sampling a random subset $R_t \subseteq \{1 \ldots t\}$ with $|R_t| = k$ such that

$$\Pr(i \in R_t) = \frac{k}{t} \qquad \forall i \in \{1 \ldots t\} \tag{7}$$

where we assume $t > k$.[4] A naive solution is to shuffle $(1 \ldots t)$ and take k positions at every step t. We can instead make incremental updates. Starting from an initial "reservoir" $z = (1 \ldots k)$, for $t = k+1, k+2, \ldots$ (1) sample $I_t \sim \text{Unif}(\{1 \ldots t\})$, (2) if $I_t \leq k$ then overwrite $z_{I_t} \leftarrow t$. For $n = 6$ and $k = 3$, this might look like

$$\begin{array}{lcr} t = 4: \; I_4 = 2 \sim \text{Unif}(\{1,2,3,4\}) & \Rightarrow & z = (1,4,3) \\ t = 5: \; I_5 = 4 \sim \text{Unif}(\{1,2,3,4,5\}) & \Rightarrow & z = (1,4,3) \\ t = 6: \; I_6 = 1 \sim \text{Unif}(\{1,2,3,4,5,6\}) & \Rightarrow & z = (6,4,3) \end{array}$$

[3] The non-uniform distribution $p_R(\{1,2\}) = p_R(\{3,4\}) = 0.5$ for $n = 4$ and $k = 2$ satisfies (5).

[4] For $t \leq k$, we can take $R_t = \{1 \ldots t\}$ and vacuously satisfy (7).

Let $R_t = \{z_1 \dots z_k\}$ at each $t > k$. Clearly, $t \in R_t$ iff $I_t \leq k$ (with probabilitiy $\frac{k}{t}$). For any $i < t$, $i \in R_t$ iff i is already in R_{t-1} and not selected. In the above example, $4 \in R_6$ because $4 \in R_5$ (which has probability $\frac{3}{5}$ by induction) is not selected (with probability $\frac{5}{6}$ since I_6 is uniform). The probability of $4 \in R_6$ is thus $\frac{1}{2}$. More generally for all $i < t$,

$$\Pr(i \in R_t) = \frac{k}{t-1} \times \frac{t-1}{t} = \frac{k}{t}$$

so (7) is satisfied. We can again work directly with the array elements $x_1, x_2, \dots$ instead of positions. The algorithm is

ReservoirSampling(k)

- $z \leftarrow ()$
- For $t = 1 \dots k$: receive x_t, append it to z.
- For $t = k+1, k+2, \dots$:
  - Receive x_t.
  - $I_t \sim \text{Unif}(\{1 \dots t\})$
  - If $I_t \leq k$: $z_{I_t} \leftarrow x_t$

At any step t, $z_1 \dots z_{\min(k,t)}$ are samples from $x_1 \dots x_t$ without replacement where every x_i is equally likely to be included.

3.1.1 Online shuffling

Without any modification, the algorithm can be used for online shuffling if the initial reservoir corresponds to a shuffle of $\{1 \dots k\}$. Let Z_t denote the reservoir after update $t > k$ where Z_{t-1} is a shuffle of $\{1 \dots t-1\}$ (i.e., $\Pr(Z_{t-1,j} = i) = \frac{1}{t-1}$ for all $j \in \{1 \dots k\}$ and $i \in \{t-1\}$). Then for all $j \in \{1 \dots k\}$,

$$\Pr(Z_{t,j} = t) = \Pr(I_t = j) = \frac{1}{t}$$

$$\forall i < t: \quad \Pr(Z_{t,j} = i) = \Pr(Z_{t-1,j} = i \text{ and } I_t \neq j) = \frac{1}{t-1} \times \frac{t-1}{t} = \frac{1}{t}$$

so Z_t corresponds to a shuffle of $\{1 \dots t\}$.

References

[1] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13**(7), 422–426.

[2] Broder, A. Z. (1997). On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE.

[3] Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, **31**(2), 249–260.

A Hash Functions

We index from 0 for convenience. An ideal **hash function** $h : \mathcal{X} \to \{0 \dots m-1\}$ satisfies

- *Deterministic*: For any $x \in \mathcal{X}$, the output $h(x) \in \{0 \dots m-1\}$ is always the same.
- *Uniform*: Assuming a distribution **pop** over $\mathcal{X}$, all m bins are used equally likely:

$$\Pr_{x \sim \mathbf{pop}}(h(x) = i) = \frac{1}{m} \qquad \forall i \in \{0 \dots m-1\} \tag{8}$$

- *Efficient*: The time and space required for hashing is constant in $|\mathcal{X}|$ and m.

We have a **collision** when $h(x) = h(y)$ for $x \neq y$. Hash functions are used to implement a **hash table**, a data structure for efficiently maintaining keys in $\mathcal{X}$.[5] But they also have direct applications in randomized algorithms.

A.1 Modular Arithmetic

To achieve binning into m buckets, hashing relies on modular arithmetic. Recall that for an integer $m \geq 1$, "$a = b \mod m$" means a is the remainder when dividing b by m (i.e., $b = km + a$ for some integer k). Equivalently, m is the divisor of their difference: $b - a = km$. Any pair of integers (a, b) whose difference is divisible by m are **congruent** modulo m, denoted by $a \equiv b \pmod{m}$. This is an equivalence relation, thus reflexive, symmetric, and transitive, and compatible with addition, subtraction, and multiplicaiton. An integer a generates the **equivalence class** modulo m by $\bar{a} = \{a + km : k \in \mathbb{Z}\}$, with the value within $\{0 \dots m-1\}$ serving as the representative. The set of all m equivalence classes modulo m forms a ring, denoted $\mathbb{Z}/m\mathbb{Z}$. It is beyond the scope of this note to cover the mathematical richness of modular arithmetic. Instead, we will use basic identities that are useful for computational purposes (e.g., Lemma B.3). One property of note is the condition for the existence of a multiplicative inverse. The **modular multiplicative inverse** of a modular m is an integer denoted $a^{-1} \in \mathbb{Z}$ such that $aa^{-1} \equiv 1 \pmod{m}$. It does not always exist (e.g., for $a = 2$ and $m = 4$). However, it exists and is unique iff a and m are coprime (i.e., $\gcd(a, m) = 1$), in particular for all $a \in \{1 \dots m-1\}$ if m is prime, allowing us to solve linear congruence like $ax \equiv b \pmod{m}$ by $x \equiv a^{-1}b \pmod{m}$. There are algorithms to find the inverse quickly, in $O(\log m)$ and much faster under mild conditions.

A.2 Hashing Integers

A common hashing scheme for $\mathcal{X} = \mathbb{Z}$ is **division hashing**. In the simplest form, we may define a family of **linear hash functions** $h_{a,b} : \mathbb{Z} \to \{0 \dots m-1\}$ indexed by $a \in \mathbb{Z}$ coprime with m and $b \in \{0 \dots m-1\}$ where

$$h_a(x) = (ax + b) \mod m \tag{9}$$

We can show that h_a cycles through some permutation π_a of $\{0 \dots m-1\}$ over $\mathbb{Z}$ (Lemma B.8). For instance, $h_a(x) = (2x + 1) \mod 3$ cycles through $(1, 0, 2)$ starting from $x = 0$. By randomly sampling the parameters a, b, we can simulate sampling a random permutation of $\{0 \dots m-1\}$.[6] This fact is useful for applications like MinHash (Section 2.1). More generally for vector-valued inputs, linear hash functions $h_{a,b} : \mathbb{Z}^d \to \{0 \dots m-1\}$ are indexed by $a \in \mathbb{Z}^d$ and $b \in \mathbb{Z}$ and compute

$$h_{a,b}(x) = (a^\top x + b) \mod m \tag{10}$$

A.3 Hashing Strings

A common hashing scheme when $\mathcal{X}$ is a set of strings over a vocabulary $\{1 \dots V\}$ $(V < m)$ is **polynomial rolling hashing**. The only parameter is a prime $p \neq m$ typically set around V. Given a string $x = (x_0 \dots x_{n-1}) \in \{1 \dots V\}^n$, the hash function $h_p(x) \in \{0 \dots m-1\}$ computes

$$h_p(x) = (x_0 + x_1 p + x_2 p^2 + \cdots + x_{n-1} p^{n-1}) \mod m \tag{11}$$

[5] A hash table uses a hash function to quickly bin $\mathcal{X}$ to m buckets then resolves any collision (which can take $O(|\mathcal{X}|)$ in the worst case under a bad hash function). It can be seen as a tradeoff between memory and time. If infinite memory is allowed, we can use the key as the index and have no collision. If infinite time is allowed, we can discard keys and search through all values.

[6] If m is prime, we can pick any non-multiple of m as a. If m is not prime, for convenience we can pick the smallest prime number $p > m$ and use $h_{a,b}(x) = ((ax + b) \mod p) \mod m$ where we can use any non-multiple of p as a. The first modulo ensures an even spread across a large space $\{0 \dots p-1\}$ and the second projects it to $\{0 \dots m-1\}$.

We can compute (11) for all prefixes $h_i = h_p(x_0 \dots x_i)$ without explicitly computing powers of p. First, we set $u_0 = 1$ and $u_i = (u_{i-1}p) \mod m$ (Corollary (B.6)). Next, we set $h_0 = x_0$ and $h_i = (h_{i-1} + x_i u_i) \mod m$ (Lemma B.7). While this takes $O(n)$ time, we have for all $i \leq j$ (Lemma B.2)

$$h_p(x_i \dots x_j)p^i = (h_j - h_{i-1}) \mod m \tag{12}$$

Thus if we precompute the modular multiplicative inverse p^{-i} (which must exist since p^i and m are coprime), we can hash any substring $x_i \dots x_j$ in $O(1)$ time by (12).

A.3.1 Rabin-Karp algorithm

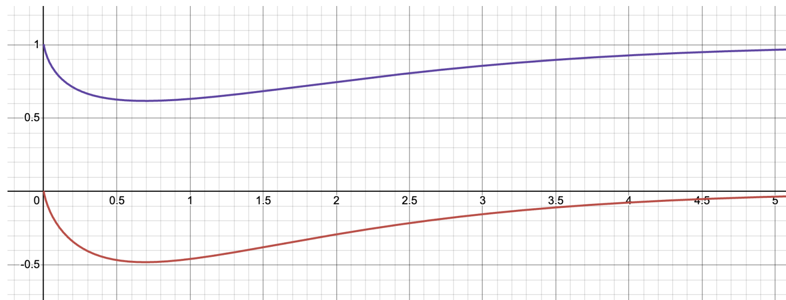
A more direct application of (12) is finding all occurrences of a string y of length n_y in a string x of length n_x in $O(n_x + n_y)$ (instead of the naive $O(n_x n_y)$). This is the **Rabin-Karp algorithm** [3]. The code below is self-explanatory. Note that it is possible to have false positives due to collisions but unlikely if m is sufficiently large. We never have false negatives; the algorithm always captures all occurrences of y in x.

```
def vocab(c): return ord(c) - ord('a') + 1  # a...z -> 1...26

p = 31  # Prime roughly equal to vocab size 26
m = int(1e9 + 9)  # Some large number > 26

def compute_hashes(x):
    hashes = [vocab(x[0])]
    for i in range(1, len(x)):
        hashes.append((hashes[-1] + vocab(x[i]) * pow(p, i, m)) % m)
    return hashes

def rabinkarp(x, y):
    h = compute_hashes(x)
    h_y = compute_hashes(y)[-1]
    occ = []
    for i in range(len(x) - len(y) + 1):
        if h_y * pow(p, i, m) % m == (h[i + len(y) - 1] - (h[i - 1] if i > 0 else 0)) % m:
            occ.append(i)
    return occ

print(rabinkarp('abcdddeebcdddaabdecdddebabcdddadcd', 'cddda'))  # [9, 26]
```

B Lemmas

Lemma B.1. The function $f : (0, \infty) \to \mathbb{R}$ defined by $f(k) := \left(1 - e^{-\frac{n}{m}k}\right)^k$ is minimized at $k^\star = (\ln 2)\frac{m}{n}$, achieving the minimum value $f^\star = 2^{-(\ln 2)\frac{m}{n}}$.

Proof. By plotting f we see that it has a unique minimum over $k \in (0, \infty)$ even though it is highly nonlinear and nonconvex (the purple line below, assuming $n = m$).



We may find the minimizer of $g(k) = \ln f(k) = k \ln(1 - e^{-\frac{n}{m}k})$ instead (the red line above). Its derivative is

$$g'(k) = \log\left(1 - e^{-\frac{n}{m}k}\right) + \frac{\left(\frac{n}{m}k\right)e^{-\frac{n}{m}k}}{1 - e^{-\frac{n}{m}k}}$$

Following the nonconvex analysis by Yuval Filmus, we use variable substitution $z := e^{-\frac{n}{m}k} \in (0,1)$ and minimize g' over z. Plugging in z and also $\left(\frac{n}{m}k\right) = -\ln z$, we have

$$g'(z) = \frac{(1-z)\ln(1-z) - z\ln z}{1-z} = \begin{cases} > 0 & \text{for all } z \in (0, \frac{1}{2}) \\ = 0 & \text{for } z = \frac{1}{2} \\ < 0 & \text{for all } z \in (\frac{1}{2}, 1) \end{cases}$$

This means that f is monotonically increasing over $0 < z < \frac{1}{2}$, stationary at $z = \frac{1}{2}$, and monotonically decreasing over $\frac{1}{2} < z < 1$. Mapping back to $k = -(\ln z)\frac{m}{n}$, equivalently we have that f is monotonically decreasing over $0 < k < (\ln 2)\frac{m}{n}$, stationary at $k = (\ln 2)\frac{m}{n}$, and monotonically increasing over $k > (\ln 2)\frac{m}{n}$. This implies that $k = (\ln 2)\frac{m}{n}$ is the unique minimizer of f over $(0, \infty)$. □

Lemma B.2. Pick any $i \leq j$ in $\{0 \dots n-1\}$. Then

$$h_p(x_i \dots x_j)p^i = (h_p(x_0 \dots x_j) - h_p(x_0 \dots x_{i-1})) \mod m$$

Proof. We have from definition (11)

$$h_p(x_i \dots x_j) = (x_i + x_{i+1}p + \dots + x_j p^{j-i}) \mod m = \left(\sum_{k=i}^{j} x_k p^{k-i}\right) \mod m$$

Thus $h_p(x_i \dots x_j) \equiv \sum_{k=i}^{j} x_k p^{k-i} \pmod{m}$. Scaling by p^i gives us $h_p(x_i \dots x_j)p^i \equiv \sum_{k=i}^{j} x_k p^k \pmod{m}$. Using (14) we have

$$\begin{aligned} h_p(x_i \dots x_j)p^i &= \left(\sum_{k=0}^{j} x_k p^k - \sum_{k=0}^{i-1} x_k p^k\right) \mod m \\ &= \left(\left(\sum_{k=0}^{j} x_k p^k \mod m\right) - \left(\sum_{k=0}^{i-1} x_k p^k \mod m\right)\right) \mod m \\ &= (h_p(x_0 \dots x_j) - h_p(x_0 \dots x_{i-1})) \mod m \end{aligned}$$

□

Lemma B.3.

$$(a+b) \mod m = \big((a \mod m) + (b \mod m)\big) \mod m \quad (13)$$

$$(a-b) \mod m = \big((a \mod m) - (b \mod m)\big) \mod m \quad (14)$$

$$(ab) \mod m = \big((a \mod m)(b \mod m)\big) \mod m \quad (15)$$

Proof. We have $\big(X \mod m\big) \equiv X \pmod{m}$ for any X. Congruence is symmetric, transitive, and compatible with addition, subtraction, and multiplication, thus

$$\begin{aligned} (a+b) \mod m &\equiv a+b \equiv \big((a \mod m) + (b \mod m)\big) \pmod{m} \\ (a-b) \mod m &\equiv a-b \equiv \big((a \mod m) - (b \mod m)\big) \pmod{m} \\ (ab) \mod m &\equiv ab \equiv \big((a \mod m)(b \mod m)\big) \pmod{m} \end{aligned}$$

□

Lemma B.4.

$$(a+b) \mod m = \big((a \mod m) + b\big) \mod m \quad (16)$$

$$(ab) \mod m = \big((a \mod m)b\big) \mod m \quad (17)$$

Proof. By applying (13) and (15) twice each:

$$\big((a \mod m) + b\big) \mod m = \big((a \mod m) + (b \mod m)\big) \mod m = \big(a + b\big) \mod m$$
$$\big((a \mod m)b\big) \mod m = \big((a \mod m)(b \mod m)\big) \mod m = (ab) \mod m$$

□

Corollary B.5 (From Lemma B.4)**.**

$$\left(\sum_{i=1}^{n} a_i\right) \mod m = \left(\cdots\big((a_1 \mod m) + a_2\big) \mod m\big) + \cdots + a_n\right) \mod m \tag{18}$$
$$\left(\prod_{i=1}^{n} a_i\right) \mod m = \left(\cdots\big((a_1 \mod m)a_2\big) \mod m\big) \cdots a_n\right) \mod m \tag{19}$$

Corollary B.6 (From (19))**.** Define $u_i = p^i \mod m$. The following procedure correctly computes u_i (without computing p^i) for all $i \in \{0, 1, \ldots, n-1\}$:

- $u_0 \leftarrow 1$
- For $i = 1 \ldots n-1$: $u_i \leftarrow (u_{i-1}p) \mod m$.

Lemma B.7. For $x = (x_0 \ldots x_{n-1})$ where $x_i < m$,

$$\left(\sum_{i=0}^{n-1} x_i p^i\right) \mod m = ((\cdots(((x_0 + x_1u_1) \mod m) + x_2u_2) \mod m + \cdots) + x_{n-1}u_{n-1}) \mod m$$

where $u_i = p^i \mod m$.

Proof. By (18),

$$\left(\sum_{i=0}^{n-1} x_i p^i\right) \mod m = ((\cdots(((x_0 + x_1p) \mod m) + x_2p^2) \mod m + \cdots) + x_{n-1}p^{n-1}) \mod m$$

The innermost term is

$$\begin{aligned}
(x_0 + x_1p) \mod m &= ((x_0 \mod m) + (x_1p \mod m)) \mod m && \text{(by (13))}\\
&= (x_0 + (x_1p \mod m)) \mod m && \text{(since } x_0 < m)\\
&= (x_0 + ((x_1 \mod m)(p \mod m) \mod m)) \mod m && \text{(by (15))}\\
&= (x_0 + (x_1u_1 \mod m)) \mod m && \text{(since } x_1 < m)\\
&= (x_0 + x_1u_1) \mod m && \text{(by (13))}
\end{aligned}$$

Similarly working through the next innermost terms, we have the statement. □

Lemma B.8. Let $h_a(x) = (ax+b) \mod m$ where a and m are coprime. Then h_a cycles through some permutation π_a of $\{0 \ldots m-1\}$ over $\mathbb{Z}$.

Proof. For simplicity, assume $b = 0$ until the end of the proof (i.e., $h_a(x) = ax \mod m$). Since a and m are coprime, there exists the (nonzero) modular multiplicative inverse a^{-1} of a. We show that h_a is 1-to-1 and onto:

- **1-to-1**: Pick any $x \neq y$ from $\{0 \ldots m-1\}$ and suppose $h_a(x) = h_a(y)$. Then

  $$\begin{aligned}
  ax \equiv ay \pmod m \quad &\Leftrightarrow \quad x \equiv y \pmod m\\
  &\Leftrightarrow \quad x = y
  \end{aligned}$$

  where the first equivalence follows by multiplication on both sides with a^{-1} and the second by the fact that $x, y \in \{0 \ldots m-1\}$. Thus it must be that $h_a(x) \neq h_a(y)$.

- **Onto**: Pick any $y \in \{0 \dots m-1\}$. Let $x = a^{-1}y \mod m \in \{0 \dots m-1\}$. Then

$$\begin{aligned}
h_a(x) &= a(a^{-1}y \mod m) \mod m \\
&= a(a^{-1}y) \mod m && \text{(by (17))} \\
&= y \mod m \\
&= y && \text{(since } y \in \{0 \dots m-1\}\text{)}
\end{aligned}$$

Thus h_a is bijective between $\{0 \dots m-1\}$ and $\{0 \dots m-1\}$ (i.e., a permutation). To show that h_a forms a cycle with period m, pick any $x \in \mathbb{Z}$ and note

$$\begin{aligned}
h_a(x+m) &= (ax + am) \mod m \\
&= ((ax \mod m) + (am \mod m)) \mod m && \text{(by (13))} \\
&= ax \mod m \\
&= h_a(x)
\end{aligned}$$

Finally, we note that using a nonzero bias term $b \neq 0$ simply shifts this cycle. $\square$