Problem 1: Bloom filters can be used to estimate set differences. Suppose Alice has a set X and Bob has a set Y, both with n elements. For example, the sets might represent their top 100 favorite songs. They both create Bloom filters of their sets, using the same number of bits m and the same k hash functions. Determine the expected number of bits where their Bloom filters differ as a function of m, n, k, and $|X \cap Y|$. Explain how this could be used as a tool to find people with the same taste in music more easily than comparing their playlists by brute force. (**Hint:** *Special cases for small values of $|X \cap Y|$ may help.*)

Solution. Consider the conditions under which the i^{th} bit of the Bloom filters differ. Certainly, (1) All common elements in $X \cap Y$ cannot be mapped to bit i. Furthermore, exactly one of the following needs to happen:

- (2.1) Some element in $X \setminus (X \cap Y)$ is mapped to bit i, but none in $Y \setminus (X \cap Y)$ is, or
- (2.2) Some element in $Y \setminus (X \cap Y)$ is mapped to bit i, but none in $X \setminus (Y \cap X)$ is.

Observe the two cases are symmetrical. Thus,

$$\mathbb{P}(i^{\text{th}} \text{ bit differ}) = \mathbb{P}((1) \text{ and } (2.1)) + \mathbb{P}((1) \text{ and } (2.2)) = 2 \cdot \mathbb{P}((1) \text{ and } (2.1))$$

so we disregard case (2.2). With k functions and m bits, each element has a probability $(1 - 1/m)^k$ of completely avoiding bit i, and so

$$\mathbb{P}((1)) = \left(1 - \frac{1}{m}\right)^{k \cdot |X \cap Y|}.$$

On the other hand,

$$\mathbb{P}(\text{nothing in } Y \setminus (X \cap Y) \text{ mapped to } i) = \left(1 - \frac{1}{m}\right)^{k(n - |X \cap Y|)}$$

and

$$\mathbb{P}(\text{something in } X \setminus (X \cap Y) \text{ mapped to bit } i) = 1 - \underbrace{\left(1 - \frac{1}{m}\right)^{k(n - |X \cap Y|)}}_{\text{nothing in } X \setminus (X \cap Y) \text{ mapped to } i}$$

so

$$\mathbb{P}((2)) = \left(1 - \frac{1}{m}\right)^{k(n - |X \cap Y|)} \left(1 - \left(1 - \frac{1}{m}\right)^{k(n - |X \cap Y|)}\right).$$

It follows that

$$\mathbb{P}(i^{\text{th}} \text{ bit differs}) = 2 \cdot \mathbb{P}((1)) \cdot \mathbb{P}((2.1)) = 2 \left(1 - \frac{1}{m}\right)^{kn} \left(1 - \left(1 - \frac{1}{m}\right)^{k(n - |X \cap Y|)}\right).$$

Finally, by linearity of expectation,

$$\mathbb{E}[\# \text{ of different bits}] = \sum_{i=1}^{m} \mathbb{P}(i^{\text{th}} \text{ bit differs}) = 2m \left(1 - \frac{1}{m}\right)^{kn} \left(1 - \left(1 - \frac{1}{m}\right)^{k(n - |X \cap Y|)}\right).$$

Problem 2: Suppose we are using an open-addressed hash table of size m to store n items, with $n \leqslant m/2$. Assume an ideal random hash function. For any i, let X_i denote the number of probes required for the i^{th} insertion into the table, and let $X = \max_i X_i$ denote the length of the longest probe sequence. (Unless otherwise specified, we use log to denote the natural logarithm.)

(i) Prove that $\Pr(X_i > k) \leqslant 1/2^k$ for all i and k.

(ii) Prove that $\Pr(X_i > 2\log n) \leqslant 1/n^2$ for all i.

(iii) Prove that $\Pr(X > 2\log n) \leqslant 1/n$. (***Hint***: *Each of the first three parts should be a one-liner, if done right.*)

(iv) Prove that $\mathbb{E}[X] = O(\log n)$. (**Hint:** *For a discrete variable,* $\mathbb{E}[X] = \sum_k k \cdot \Pr(X = k)$. *Split the sum into two parts where one behaves "nicely" and the other sufficiently small.*)

Solution.

(i) A one-liner from Nick Maroulis: Because $n \leqslant m/2$, never more than half of the table is occupied, so each collision occurs with probability $\leqslant 1/2$, meaning k collisions (i.e. $X_i > k$) occur with probability $\leqslant (1/2)^k$.

(ii) Plug $k = 2\log n$ into the previous part and we get $\mathbb{P}(X_i > 2\log n) \leqslant 1/2^{2\log n} = 1/n^2$.

(iii) If $X > 2\log n$ then some $X_i > 2\log n$. Union bound implies

$$\mathbb{P}(\text{some } X_i > 2\log n) \leqslant \sum_{i=1}^{n} \mathbb{P}(X_i > 2\log n) \leqslant \frac{n}{n^2} = \frac{1}{n}.$$

(iv) From Nick Maroulis: we can split $\mathbb{E}[X]$ into two parts:

- $X > 2\log n$. This happens with probability $\leqslant 1/n$, and when this happens, we still know $X \leqslant m$.
- $X \leqslant 2\log n$. This happens with probability $\leqslant 1$ just by definition of probability, and when this happens, $X \leqslant 2\log n$ (almost sounds like tautology).

Therefore,

$$\begin{aligned}\mathbb{E}[X] &\leqslant \mathbb{P}(X \leqslant 2\log n)(2\log n) + \mathbb{P}(X > 2\log n)(m) \\ &\leqslant 2\log n + \frac{m}{n} \leqslant 2\log n + 2 = \mathcal{O}(\log n).\end{aligned}$$

Problem 3: Let $a = \langle a_0, \ldots, a_{n-1} \rangle$ and $b = \langle b_0, \ldots, b_{n-1} \rangle$ be two vectors, each of length n with nonnegative elements. Let $[n] = \{0, 1, \ldots, n-1\}$. We wish to perform two operations on a and b:

(i) Update (x, i, Δ): For $x \in \{a, b\}$, $i \in [n]$, and $\Delta > 0$, set $x_i \leftarrow x_i + \Delta$.

(ii) Product (a, b): Estimate $a \odot b = \sum_{i=1}^{n} a_i \times b_i$.

Describe a data structure of size $O(\varepsilon^{-1} \log \delta^{-1})$ that estimates $a \odot b$ within error $\varepsilon \|a\|_1 \cdot \|b\|_1$, i.e.,

$$|\text{Product}(a, b) - a \odot b| < \varepsilon \|a\|_1 \cdot \|b\|_1 \tag{1}$$

with probability at least $1 - \delta$, in $O(\varepsilon^{-1} \log \delta^{-1})$ time. (Recall that for a vector $x = (x_1, \ldots, x_n)$, $\|x\|_1 = \sum_{i=1}^{n} |x_i|$.)

Describe how the two operations will be performed. Justify that your approach satisfies the error bound stated above in equation (1), and analyze the running time of each operation. (***Hint**: Two Count-Min sketches.*)

Solution. We will use one CMS on eac vector — call them T_a and T_b — with standard parameters $w = \lceil e/\epsilon \rceil$ and $d = \lceil \log(1/\delta) \rceil$. We use d hash functions, each mapping $\{1, \ldots, n\}$ to $\{1, \ldots, w\}$. The update procedure is exactly the same. For an update (x, i, Δ), we add Δ to each entry $T_x[j, h_j(i)]$ for each hash function h_j.

The key intuition is that each row-wise inner product (e.g. the dot product between the first rows of T_a, T_b) is an overestimate of the actual inner product $a \cdot b$. Therefore, so is the minimum across all rows. We define the Product operations as the minimum across all d estimates:

$$\text{Product}(a, b) = \min_{1 \leqslant j \leqslant d} \sum_{i=0}^{w-1} T_a[j, h_j(i)] \cdot T_b[j, h_j(i)].$$

Now we fix some $j \in [0, d]$ and consider $\text{Product}_j(a, b)$, the inner product between the j^{th} rows of T_a and T_b. Observe the overestimate comes solely from hash collisions:

$$\text{Product}_j(a, b) - a \cdot b = \sum_{\substack{p \neq q \\ h_j(p) = h_j(q)}} a_p b_q$$

so taking expectation gives

$$\begin{aligned} \mathbb{E}[\text{row } j\text{'s overestimate}] &= \sum_{p \neq q} \mathbb{P}(h_j(p) = h_j(q)) \cdot a_p b_q \\ &= \sum_{p \neq q} \frac{1}{w} \cdot a_p b_q \leqslant \frac{1}{w} \sum_{p,q} a_p b_q = \frac{\epsilon}{e} \|a\|_1 \|b\|_1. \end{aligned}$$

We just need one more application of Markov's inequality to conclude the proof:

$$\mathbb{P}(\text{row } j\text{'s overestimate} > \epsilon \|a\|_1 \|b\|_1) \leqslant \frac{\mathbb{E}[\text{row } j\text{'s estimate}]}{\epsilon \|a\|_1 \|b\|_1} = \frac{1}{e}$$

so

$$\mathbb{P}(\text{all rows overestimate by} > \epsilon \|a\|_1 \|b\|_1) \leqslant \left(\frac{1}{e}\right)^d = \delta.$$

Problem 4: Suppose you are given a document of n characters from an alphabet of m characters. Assume $n \gg m$ (for example, $n \in [10^6, 10^7]$ for novels such as *Count of Monte Cristo* and *A Tale of Two Cities*, but $m < 100$). You are also given the frequencies $p_1, p_2, \ldots, p_m$ with which the m characters appear.

(i) Suppose we want to compress the document. We first build the Huffman code tree and then encode the document using that tree. What difference does it make to the overall running time if we use an unsorted array implementation of a priority queue versus a binary heap implementation of a priority queue when building the Huffman code tree? State the total runtime complexity of building the tree and encoding the document for both cases as a function of n and m, and comment on the significance of the difference under the assumption that $n \gg m$.

(ii) As mentioned in the lecture, encoding a document directly from the Huffman code tree (without using a lookup table) could take $O(nm)$ time in the worst case. Why is it $O(nm)$ instead of $O(n \log(m))$, even though the code tree is a binary tree?

(iii) Consider using the Huffman code tree directly to decode. Would you expect this to be faster or slower than encoding (again directly using the Huffman code tree)? Why?

Solution.

(i) With an unsorted array implementation, it takes linear time to extract the two minimum of an array, and repeating this process till the array has one single element will take $\mathcal{O}(m^2)$ time total. Lookup is $\mathcal{O}(1)$ for each character, and for n total characters this takes $\mathcal{O}(n)$. Therefore, the total complexity is $\mathcal{O}(m^2 + n)$.

With a binary heap, building the tree will take $\mathcal{O}(\log m)$ time for each extract-min and $\mathcal{O}(m \log m)$ total. Lookup remains $\mathcal{O}(n)$ for the entire document, so the total complexity is now $\mathcal{O}(m \log m + n)$.

When $n \gg m$, the different runtime caused by different data structure becomes negligible, as the true bottleneck lies in decoding the document, the $\mathcal{O}(n)$ term.

(ii) The worse case for lookup is $\mathcal{O}(mn)$ even with binary tree because the tree can be extremely unbalanced, essentially resembling a chain of height/length $\mathcal{O}(m)$.

(iii) One would expect the decoding process to be faster in practice, as decoding performs simpler operations using predetermined paths, while the encoding process involves computing these paths real-time.