Due date: February 04, 2025

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.*)

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.*)

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.*)

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?

You will need to implement at least three functionalities. In addition, feel free to implement any relevant helper functions or additional classes. You are responsible for deciding how to structure and test your code as you develop.

- **BuildHuffmanCodeTree.** Takes an alphabet represented as an array of characters and an array of integer counts corresponding to those characters, and produces the greedy Huffman encoding tree.[1] For example, an alphabet might consist of [a, b, c] with corresponding counts [1, 2, 3] denoting that a occurs 1 time in the document to compress, b occurs 2 times, and c occurs 3 times. Tip: You may find it helpful to implement a Huffman class in order to organize your code.
- **Encode.** Takes a Huffman encoding tree and a message represented as a string of characters in the alphabet of the Huffman encoding tree, and generates a string of 1s and 0s[2] representing the message according to the Huffman encoding tree. For example, a message on the alphabet [a, b, c] might consist of abbccc and with the frequencies [1, 2, 3], a valid Huffman encoding might be 101111000 where 10 is for a, 11 is for b, and 0 is for c.
- **Decode.** Takes a Huffman encoding tree and string of 1s and 0s, and generates a message as a string of characters from the alphabet of the Huffman encoding tree. For example, given the previous encoded message 101111000 and the encoding tree, this should recover the message abbccc.

[1] You are welcome to use or lookup standard library implementations of binary trees or priority queues. If you write your own binary tree or priority queue code, you are not required to optimize your implementations (for example, you can implement your priority queue as an unsorted array with $O(m)$ extractMin() instead of $O(\log(m))$ as in a binary heap implementation).

[2] Note that bits are an actual type in many languages, but you do not need to use this type for your homework (or to understand the encoding more generally). You can just use a standard string in Python, Java, or your preferred language, and represent 1s and 0s by the corresponding characters.