This homework contains three problems. Read the instructions for submitting homework on the course webpage.

Collaboration Policy: For this homework, each student can work in a group with up to three members. Only one solution for each group needs to be submitted. Follow the submission instructions carefully.

1. Your boss wants you to find a perfect hash function for mapping a known set of $n$ items into a table of size $m$. A hash function is perfect if there are no collisions; each of the $n$ items is mapped to a different slot in the hash table. Of course, a perfect hash function is only possible if $m \geq n$. After cursing your algorithms instructor for not teaching you about (this kind of) perfect hashing, you decide to try something simple: repeatedly pick ideal random hash functions until you find one that happens to be perfect.

Ideal randomness means that the hash function is chosen uniformly at random from the set of all functions from $U$ to $\{0, 1, \ldots, m-1\}$. Intuitively, an ideal random hash function is a function $h : U \rightarrow \{0, 1, \ldots, m-1\}$ such that for each $x \in U$ the value of $h(x)$ is decided by rolling an unbiased $m$-sided die.

(a) Suppose you pick an ideal random hash function $h$. What is the exact expected number of pair-wise collisions, as a function of $n$ (the number of items) and $m$ (the size of the table)? Don’t worry about how to resolve collisions; just count them. For $x \neq y$, if $h(x) = h(y)$ then we say that $(x, y)$ constitute a pair-wise collision.

(b) What is the exact probability that a random hash function is perfect?

(c) What is the exact expected number of different random hash functions you have to test before you find a perfect hash function?

(d) What is the exact probability that none of the first $N$ random hash functions you try is perfect?

(e) How many ideal random hash functions do you have to test to find a perfect hash function with high probability?

2. Tabulated hashing uses tables of random numbers to compute hash values. Suppose $|U| = 2^w \times 2^w$ and $m = 2^l$, so that the items being hashed are pairs $(x, y)$ where $x$ and $y$ are $w$-bit strings (or $2w$-bit strings broken in half), and hash values are $l$-bit strings.

Let $A[0 \cdots 2^w - 1]$ and $B[0 \cdots 2^w - 1]$ be arrays of $l$-bit strings ($A$ and $B$ can be though of as $2^w \times l$ dimensional array of bits). Define the hash function $h_{A,B} : U \rightarrow [m]$ by setting

$$h_{A,B}(x, y) := A[x] \oplus B[y]$$

where $\oplus$ denotes bit-wise exclusive-or. Let $\mathcal{H}'$ denote the set of all possible functions $h_{A,B}$. Note that sampling an $h_{A,B} \in \mathcal{H}'$ uniformly at random is equivalent to setting every bit of the arrays $A$ and $B$ to 0 or 1 uniformly at random.
For an integer $k > 0$, we say that a family of hash functions $\mathcal{H}$ mapping $U$ to $\{0, 1, \cdots, (m-1)\}$ is $k$-uniform if for any sequence of $k$ disjoint keys and any sequence of $k$ hash values, the probability that each key maps to the corresponding hash value is $\frac{1}{m^k}$

$$\Pr_{h \sim \mathcal{H}} \left[ \bigwedge_{j=1}^{k} h(x_j) = i_j \right] = \frac{1}{m^k} \quad \text{for all distinct } x_1, \cdots x_k \in U, \text{ and all } i_1, \cdots i_k \in \{0, \ldots, (m-1)\}$$

In the above, $h \sim \mathcal{H}$ means function $h$ is picked uniformly at random from family $\mathcal{H}$. (For more details on $k$-uniform family of hash functions, see Jeff’s notes (page 3): [https://courses.engr.illinois.edu/cs473/sp2016/notes/12-hashing.pdf](https://courses.engr.illinois.edu/cs473/sp2016/notes/12-hashing.pdf))

(a) Prove that $\mathcal{H}'$ is 2-uniform.

(b) Prove that $\mathcal{H}'$ is 3-uniform. [Hint: Solve part (a) first.]

(c) Prove that $\mathcal{H}'$ is not 4-uniform.

Yes, “see part (b)” is worth full credit for part (a), but only if your solution to part (b) is correct.

3. In lecture we discussed the Karp-Rabin randomized algorithm for pattern matching. The power of randomization is seen by considering the two-dimensional pattern matching problem. The input consists of an arbitrary $n \times n$ binary matrix $T$ and an arbitrary $m \times m$ binary matrix $P$, where $m < n$. Our goal is to check if $P$ occurs as a (contiguous) submatrix of $T$. Describe an algorithm that runs in $O(n^2)$ time assuming that arithmetic operation in $O(\log n)$-bit integers can be performed in constant time. This can be done via a modification of the Karp-Rabin algorithm. To achieve this, you will have to apply some ingenuity in figuring out how to update the fingerprint in only constant time for most positions in the array.

[Hint: we can view an $m \times m$ matrix as an $m^2$-bit integer. Rather than computing its fingerprint directly, compute instead a fingerprint for each row first, and maintain these fingerprints as you move around.]