1. **Half-priced hash.** In class, we studied a family of hash functions based on taking inner products. That family was a family of hash functions from $\mathbb{Z}_m^t$ to $\mathbb{Z}_m$ indexed by $\mathbb{Z}_m^t$. For each hash function index $\lambda = (\lambda_1, \ldots, \lambda_t) \in \mathbb{Z}_m^t$, and each key $a = (a_1, \ldots, a_t) \in \mathbb{Z}_m^t$, the hash function was defined as $h_\lambda(a) := \sum a_i \lambda_i$, which requires $t$ multiplications to evaluate. In this problem, you are to analyze a variant hash which cuts the number of multiplications in half.

Assume $t$ is even, so $t = 2s$. Hash function indices and keys have the same structure as above, but the hash function is defined as follows:

$$h_\lambda'(a) := \sum_{i=1}^s (a_{2i-1} + \lambda_{2i-1})(a_{2i} + \lambda_{2i}).$$

So, for example, for $t = 4$, we have

$$h_\lambda'(a) = (a_1 + \lambda_1)(a_2 + \lambda_2) + (a_3 + \lambda_3)(a_4 + \lambda_4).$$

Your task is to show that the family of hash functions $\{h_\lambda'\}_{\lambda \in \mathbb{Z}_m^t}$ is a universal family.

Hint: Your proof should mimic the one given in class for the inner-product based family. Namely, consider two distinct keys $a = (a_1, \ldots, a_t)$ and $b = (b_1, \ldots, b_t)$, and show that the number of hash function indices $(\lambda_1, \ldots, \lambda_t)$ which satisfy

$$\sum_{i=1}^s (a_{2i-1} + \lambda_{2i-1})(a_{2i} + \lambda_{2i}) = \sum_{i=1}^s (b_{2i-1} + \lambda_{2i-1})(b_{2i} + \lambda_{2i}).$$

is at most $m^{t-1}$. To keep the notation simple, you may first want to do the calculation for the case $t = 4$.

2. **Mod-free hash.** In class, most of the hash functions we looked at required arithmetic mod $m$, where $m$ was a prime. This exercise looks at a family of hash functions where this is not necessary, which can result in a significantly more efficient implementation.

Let $k$, $\ell$, and $t$ be positive integers. Let $U := [0 \ldots 2^k]^t$, $V := [0 \ldots 2^\ell]^t$, and $A := [0 \ldots 2^{k+\ell}]^t$. We define a family of hash functions $\{h_\lambda\}_{\lambda \in A}$ from $U$ to $V$ as follows. For $a = (a_1, \ldots, a_t) \in U$ and $\lambda = (\lambda_1, \ldots, \lambda_t) \in A$, we define

$$h_\lambda(a) := \left\lfloor (a_1 \lambda_1 + \cdots + a_t \lambda_t) \bmod 2^{k+\ell} \right\rfloor / 2^\ell.$$

To see why this can be efficiently implemented, suppose that $k = 10$ and $\ell = 20$. Then using 32-bit unsigned arithmetic, we can compute $\text{sum} \leftarrow \sum a_i \lambda_i$ using $t$ integer multiplications and additions, and we can then compute the hash as $\text{hash} \leftarrow (\text{sum} \& (2^{30} - 1)) > 10$. Make sure you understand why this works before proceeding.

Your goal is to show that that $\{h_\lambda\}_{\lambda \in A}$ is a $2^{(-\ell+1)}$-universal family of hash functions.

Here is an outline you should follow:

(a) Suppose $h_\lambda(a) = h_\lambda(b)$, where $a = (a_1, \ldots, a_t)$, $b = (b_1, \ldots, b_t)$, and $\lambda = (\lambda_1, \ldots, \lambda_t)$. Show that we must have

$$c_1 \lambda_1 + \cdots + c_t \lambda_t \equiv d \pmod{2^{k+\ell}}, \quad (*)$$

where $c_i := a_i - b_i$ for $i = 1, \ldots, t$, and $|d| < 2^k$.

(b) Further suppose that $a \neq b$, so that $c_i \neq 0$ for some $i = 1, \ldots, t$. By re-ordering indices, we may assume that $c_1 = 2^j f$, where $f$ is odd and $0 \leq j < k$, and moreover, $2^j | c_i$ for $i = 2, \ldots, t$. Explain why we may assume this.

(c) Argue that we must have $2^j | d$, and therefore, that $d$ belongs to a set $S$ of size at most $2^{k-j+1}$ possible integers (you should describe the set $S$).

(d) Argue that for every choice of $d \in S$, and every choice of $\lambda_2, \ldots, \lambda_t \in [0 \ldots 2^{k+\ell})$, there are exactly $2^j$ choices of $\lambda_1 \in [0 \ldots 2^{k+\ell})$ that satisfy the congruence $(*)$. 

Basic Algorithms — Fall 2017 — Problem Set 8 (updated)

Due: Thursday, Dec 14

Note: No late assignments accepted
(e) Conclude that \( \{h_\lambda\}_{\lambda \in \Lambda} \) is a \( 2^{-t+1} \)-universal family of hash functions.

3. **Pretty good hash.** Let \( h : U \rightarrow V \) be a hash function, mapping from some (finite) universe \( U \) of keys to a (finite) set of slots \( V \). For a set \( Q \subseteq U \) and an element \( a \in Q \), we say that \( h \) **isolates** \( a \) in \( Q \) if the only element of \( Q \) that hashes to the slot \( h(a) \) is \( a \) itself, i.e.,

\[
\text{for all } b \in Q : \quad h(a) = h(b) \implies a = b.
\]

Now recall the notion of a **perfect** hash function. Using the above terminology, we can say that \( h \) is a perfect hash function for \( Q \) if \( h \) isolates every element of \( Q \). Consider the following, weaker property: let us say that \( h \) is a **pretty good hash function for** \( Q \) if \( h \) isolates at least \( |Q|/2 \) elements of \( Q \).

Your task is to design an efficient, probabilistic algorithm that takes as input a set \( Q = \{a_1, \ldots, a_n\} \) of \( n \) distinct keys, and finds a hash function that is pretty good for \( Q \).

To this end, assume that \( \{h_\lambda\}_{\lambda \in \Lambda} \) is a universal family of hash functions from \( U \) to \( V \). Assume that \( V = \{0 \ldots m\} \), where \( 4n \leq m \leq 8m \). You may assume that you can choose \( \lambda \in \Lambda \) uniformly at random in time \( O(1) \), and that you can evaluate \( h_\lambda(a) \) at any point \( a \in U \) in time \( O(1) \).

On input \( Q \) as above, your algorithm should find \( \lambda \in \Lambda \) such that \( h_\lambda \) is pretty good for \( Q \). The expected running time of your algorithm should be \( O(n) \).

You may wish to follow the following outline:

(a) Suppose \( R \) is uniformly distributed over \( \Lambda \) Let \( X \) be the number of \( a_i \)’s that are not isolated by \( h_R \). Show that \( E[X] \leq n(n-1)/m \).

Hint: use indicator variables and linearity of expectation.

(b) Now use Markov’s inequality and the assumption that \( m \geq 4n \), to show that a random hash function \( h_R \) is pretty good with probability at least \( 1/2 \).

(c) Using part (b), and the assumption that \( m \leq 8n \), design an algorithm that actually finds a pretty good hash function in expected time \( O(n) \).

4. **2D hash.** In class, we presented a \( (t-1)/m \)-universal hash family based on polynomial evaluation. This exercise develops a two-dimensional variant. The universe of keys \( U \) consists of all \( t \times t \) matrices over \( \mathbb{Z}_m \), where \( m \) is prime. We write such a matrix \( A \in U \) as \( A = (a_{ij}) \), where the indices \( i \) and \( j \) run from 0 to \( t-1 \). The set of hash function indices \( \Lambda \) consists of pairs \( (\lambda_1, \lambda_2) \in \mathbb{Z}_m \times \mathbb{Z}_m \). For \( \lambda = (\lambda_1, \lambda_2) \in \Lambda \) and \( A = (a_{ij}) \in U \), define

\[
h_\lambda(A) = \sum_{i=0}^{t-1} \sum_{j=0}^{t-1} a_{ij} \lambda_1^i \lambda_2^j. \tag{1}
\]

Show that \( \{h_\lambda\}_{\lambda \in \Lambda} \) is \( 2(t-1)/m \)-universal.

Hint: re-write the right-hand side of (1) as

\[
\sum_{i=0}^{t-1} \lambda_1^i \left( \sum_{j=0}^{t-1} a_{ij} \lambda_2^j \right),
\]

and make use of fact that any polynomial \( f(X) \) of degree at most \( t-1 \) over \( \mathbb{Z}_m \) has at most \( t-1 \) roots. You will need to use this fact twice: once with

\[
f(X) = \sum_{j=0}^{t-1} a_{ij} \lambda_1^j X^j \quad \text{for some } i \in \{0 \ldots t\},
\]

and once with

\[
f(X) = \sum_{i=0}^{t-1} X^i \left( \sum_{j=0}^{t-1} a_{ij} \lambda_2^j \right) \quad \text{for some } \lambda_2 \in \mathbb{Z}_m.
\]

Try to make your argument as precise as possible, using the law of total probability.
5. **2D pattern matching.** In the 2D pattern matching problem, you are given an \( n \times n \) array \( A \) and a \( t \times t \) array \( B \), where \( t \leq n \), and you want to determine if \( B \) appears as a subarray within \( A \). For example, the array
\[
B = \begin{pmatrix}
1 & 2 \\
3 & 4
\end{pmatrix}
\]
appears as a subarray of
\[
A = \begin{pmatrix}
0 & 1 & 1 & 0 \\
0 & 1 & 2 & 3 \\
2 & 3 & 4 & 0 \\
0 & 1 & 0 & 1
\end{pmatrix}.
\]
Adapt the Karp/Rabin pattern matching algorithm using the 2D hash function from the previous exercise to give a probabilistic algorithm that solves this problem. The expected running time should be \( O(n^2 + n^2 t^3/m) \), where \( m \) is the prime used in the above hash function. For reasonable choices of \( t \) and \( m \), the first term will dominate, and so the expected running time will be \( O(n^2) \).

Hint: you will have to somehow adapt the “rolling hash” idea of Karp/Rabin to the 2D hash.

6. **Hash 'til you crash.** Assume \( |U| > m \), and consider fixed, distinct keys \( a_1, \ldots, a_{m+1} \in U \). Suppose a hash function \( h: U \to \{0 \ldots m\} \) is chosen at random from some family of hash functions. Let \( X \) be the least positive integer \( i \) such that \( h \) maps two items among \( a_1, \ldots, a_i \) to the same slot; that is, if we insert \( a_1, \ldots, a_{m+1} \) one at a time into an initially empty hash table, then \( X \) represents the number of insertions we perform until some slot contains 2 items.

Your goal is to show that under the uniform hashing assumption, \( E[X] = O(m^{1/2}) \).

Note: Recall that under the uniform hashing assumption, the family of random variables \( \{h(a_i)\}_{i=1}^{m+1} \) is mutually independent, with each \( h(a_i) \) uniformly distributed over \( \{0 \ldots m\} \).

(a) Show that for \( j = 1, \ldots, m+1 \), we have
\[
\Pr[X \geq j] = \prod_{i=1}^{j-1} \left(1 - \frac{i - 1}{m}\right).
\]
(b) Using the part (a), along with the handy inequality \( e^x \geq 1 + x \) (which holds for all real numbers \( x \)), show that
\[
\Pr[X \geq j] \leq e^{-(j-2)^2/2m}
\]
for all \( j \geq 2 \).
(c) Using part (b), along with the tail sum formula for expectation, show that
\[
E[X] \leq 1 + \sum_{i=0}^{\infty} e^{-i^2/2m}.
\]
(d) Now use part (c), and approximate a sum by an integral, to show that
\[
E[X] = O(m^{1/2}).
\]
To do this, you may use the fact that \( \int_0^{\infty} e^{-u^2} du = \sqrt{\pi}/2 \). You may also have to recall the substitution method for integration.

7. **Breaking bag.** A bag is an abstract data type which may hold some number of distinct items. You may query the bag to determine how many items are in the bag. This takes constant time. You may also apply the probabilistic operation \( \text{split} \) to a bag, which partitions the items in the given bag into two new bags. All you know about \( \text{split} \) is that for every pair of distinct items in the input bag, the probability that both items end up in the same output bag is \( \leq 1/2 \). Also, if a bag contains \( n \) items, applying \( \text{split} \) to the bag takes time \( O(n) \).

Your goal is to design and analyze a probabilistic algorithm that takes as input a bag containing \( n \) items \( i_1, \ldots, i_n \), and produces as output \( n \) bags \( B_1, \ldots, B_n \) such that bag \( B_j \) contains the single item \( i_j \). The ordering of the output bags is irrelevant. Your algorithm should run in expected time \( O(n \log n) \).

Here is an outline you should follow:
(a) The algorithm is the obvious divide and conquer algorithm, using the split operation to get two sub-problems, and then recursing on both, as necessary. Write out this algorithm.

(b) To analyze the running time, consider any two items in the original bag, and argue that after \(d\) levels of recursion, the probability that they remain in the same bag is at most \(2^{-d}\).

(c) Using (b), argue that for any particular item in the original bag, the probability that it is not in a bag by itself after \(d\) levels of recursion is at most \(2^{-d}(n-1)\). Hint: union bound.

(d) Using (c), show that for any particular item \(i\) in the original bag, if \(D_i\) is the depth in the recursion tree at which \(i\) ends up in a bag by itself, then \(E[D_i] = O(\log n)\). Hint: use the tail sum formula \(E[D_i] = \sum_{d \geq 1} \Pr[D_i \geq d]\).

(e) Finally, argue that the running time \(T\) is \(O(\sum_i (D_i + 1))\), where the sum is over all items \(i\) in the original bag, and from this and part (d), argue that \(E[T] = O(n \log n)\). Hint: linearity of expectation.

8. **Nuts and bolts.** You have a mixed pile of \(n\) nuts and \(n\) bolts and need to quickly find the corresponding pairs of nuts and bolts. Each nut matches exactly one bolt, and each bolt matches exactly one nut. By fitting a nut and bolt together, you can see which is bigger. However, it is not possible to directly compare two nuts or two bolts. Design and analyze an probabilistic algorithm for this problem with an \(O(n \log n)\) expected running time.

*Hint:* Customize QuickSort to the problem.

*Note:* Only a very complicated deterministic \(O(n \log n)\) algorithm is known for this problem.