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 $h_\lambda(a) := \sum_{i=1}^{t} a_i \lambda_i$, which requires $t$ multiplications to evaluate. In this problem, you are to analyze a variant hash which cuts the number of multiplications in half. In most computing environments, multiplications are much more expensive than addition, so this can speed up the hash function evaluation significantly.

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

$$h_{\lambda}(a) := \left(\sum_{i=1}^{t} a_i \lambda_i \right) \mod 2^{k+\ell} / 2^{\ell}.$$

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

Your goal is to show that that $\{h_{\lambda}\}_{\lambda \in \Lambda}$ 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 $c_1 \equiv 2^j f$, where $f$ is odd and $0 \leq j < k$, and moreover, $2^j \mid c_i$ for $i = 2, \ldots, t$. Explain why we may assume this.

(c) Argue that we must have $2^j \mid d$, 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 $(*)$. 

Due: Thursday morning, May 3
3. **Pretty good hash.** Let $h : \mathcal{U} \to \mathcal{V}$ be a hash function, mapping from some (finite) universe $\mathcal{U}$ of keys to a (finite) set of slots $\mathcal{V}$. For a set $Q \subseteq \mathcal{U}$ 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 $\mathcal{U}$ to $\mathcal{V}$. Assume that $\mathcal{V} = \{0 \ldots m\}$, where $4n \leq m \leq 8n$. 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 $a \in \mathcal{U}$ 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. **Composed hash.** Suppose $\{h_\lambda\}_{\lambda \in \Lambda}$ is an $\epsilon$-universal family of hash functions from $\mathcal{U}$ to $\mathcal{V}$. Further, suppose that $\{h'_\lambda\}_{\lambda' \in \Lambda'}$ is an $\epsilon'$-universal family of hash functions from $\mathcal{V}$ to $\mathcal{W}$. Your task is to show that

$$\{h'_\lambda \circ h_\lambda\}_{(\lambda, \lambda') \in \Lambda \times \Lambda'}$$

is an $(\epsilon + \epsilon')$-universal family of hash functions from $\mathcal{U}$ to $\mathcal{W}$.

Here, $h'_\lambda \circ h_\lambda$ is the usual composition of functions $h'_\lambda$ and $h_\lambda$, so that $(h'_\lambda \circ h_\lambda)(a) = h'_\lambda(h_\lambda(a))$. That is, to hash a key $a \in \mathcal{U}$ using the hash function $h'_\lambda \circ h_\lambda$, we first compute the intermediate value $s := h_\lambda(a)$, and then we compute the result $t := h'_\lambda(s)$.

To prove this result, you should show that for any given pair of keys $a, b \in \mathcal{U}$ with $a \neq b$, the number of pairs $(\lambda, \lambda') \in \Lambda \times \Lambda'$ such that $h'_\lambda(h_\lambda(a)) = h'_\lambda(h_\lambda(b))$ is at most $(\epsilon + \epsilon')|\Lambda| \cdot |\Lambda|$. To this end, separately calculate upper bounds on

(i) the number of indices $\lambda \in \Lambda$ such that $h_\lambda(a) = h_\lambda(b)$;

(ii) the number of pairs $(\lambda, \lambda') \in \Lambda \times \Lambda'$ such that $h_\lambda(a) = h_\lambda(b)$ but $h'_\lambda(h_\lambda(a)) = h'_\lambda(h_\lambda(b))$.

For (i), use the fact that $\{h_\lambda\}_{\lambda \in \Lambda}$ is $\epsilon$-universal; for (ii), use the fact that $\{h'_\lambda\}_{\lambda' \in \Lambda'}$ is $\epsilon'$-universal.

5. **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 $\mathcal{U}$ consists of all $t \times t$ matrices over $\mathbb{Z}_m$, where $m$ is prime. We write such a matrix $A \in \mathcal{U}$ 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 $A = (a_{ij}) \in \mathcal{U}$, define

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

(1)

Show that $\{h_\lambda\}_{\lambda \in \Lambda}$ is $2(t - 1)/m$-universal.
Hint: re-write the right-hand side of (1) as
\[ \sum_{i=0}^{t-1} \lambda_i \left( \sum_{j=0}^{t-1} a_{ij} \lambda_j \right), \]
and then apply the result of the previous exercise (with \( V = W = \mathbb{Z}_m^t \)), making use of the fact that the usual polynomial evaluation hash is \( (t-1)/m \)-universal.

6. **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.

7. **Birthday paradox.** Consider fixed, distinct keys \( a_1, \ldots, a_n \in \mathcal{U} \). Suppose a hash function \( h: \mathcal{U} \to [0..m) \) is chosen at random from some family of hash functions, where \( m \geq n \geq 2 \). Let \( p_{n,m} \) be the probability that \( h(a_i) = h(a_j) \) for some \( i \neq j \). That is, \( p_{n,m} \) is the probability that hashing \( n \) keys into \( m \) slots results in a collision.

We shall analyze the value \( p_{n,m} \) under the uniform hashing assumption. Recall that this means that we are assuming that the family of random variables \( \{h(a_i)\}_{i=1}^n \) is mutually independent, with each \( h(a_i) \) uniformly distributed over \( [0..m) \).

(a) Show that
\[ 1 - p_{n,m} = \prod_{i=1}^{n} \left( 1 - \frac{i-1}{m} \right). \]

(b) Using the part (a), along with the handy inequality \( 1 + x \leq e^x \) (which holds for all real numbers \( x \)), show that
\[ p_{n,m} \geq 1 - e^{-n(n-1)/2m}. \]

(c) Using part (b), show that if \( n \geq \sqrt{2\ln(2) m} + 1 \), then \( p_{n,m} \geq 1/2 \).

**Note:** This says we only have to hash \( O(\sqrt{m}) \) keys in order for there to be a collision with probability at least \( 1/2 \). This is a special case of the “birthday paradox”, which says that if there are \( 23 \) people in a room, it is more likely than not that two people in the room share the same birthday (you can plug \( n = 23 \) and \( m = 365 \) directly into the inequality in part (b) to see this).

Also note that in class, we showed that the collision probability \( p_{n,m} \) is at most \( n(n-1)/2m \), assuming \( h \) is chosen from a universal family of hash functions. This same argument works under the uniform hash assumption, so under this assumption we get both lower and upper bounds on \( p_{n,m} \).

8. **Breaking bag.** A bag is an abstract data type which may hold any number of objects. You may query the bag to determine how many objects are in the bag. This takes constant time. You may also apply the probabilistic operation \( \text{split} \) to a bag, which partitions the objects in the given bag into two new bags. All you know about \( \text{split} \) is that for every pair of objects in the input bag, the probability that both objects end up in the same output bag is \( \leq 1/2 \). Also, if a bag contains \( n \) objects, 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 $n$ objects and produces as output $n$ bags such that each output bag contains a single object. 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 subproblems, and then recursing on both, as necessary. Write out this algorithm.

(b) To analyze the running time, consider any two objects 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 object 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 object $i$ 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 objects $i$ in the original bag, and from this and part (d), argue that $E[T] = O(n \log n)$. Hint: linearity of expectation.

9. **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.