1 Probability Review

- **Expectation** (weighted average): the expectation of a random quantity X is:

$$\sum_{x=-\infty}^{\infty} x \cdot P(X = x)$$

  For each value x that X can take on, we look at the probability X will take on that value and then multiply it by that value. This is the average value of X because each possible outcome x is weighted by its probability of occurring.

- **Independence**: If two events are independent, then the probability that both will happen is the product of the probabilities that each will happen.

- **Complement**: Sometimes, if it is difficult to calculate the probability of something happening, you can calculate the probability of it *not* happening, and then subtract that from 1.

- **Linearity of Expectation:** When trying to calculate the expected number of A's that have property B, you simply find for each A, what is the probability that property B holds for it, and sum that probability over all A's.

- **Recursion:** In some problems, you can write down a recursive formula for the expectation of a quantity. Solving the recursive formula will yield the numerical answer.

Exercise 1. An unfair coin comes up heads with probability $\frac{1}{3}$ and tails with probability $\frac{2}{3}$.

(a) If I flip the coin 3 times, what's the expected number of times it comes up as heads? Why should or should this not be 1.5?

(b) Perform the calculation from (a) again, but this time using linearity of expectation. How many heads should you expect if you flip the coin 10 times?

(c) On average, how many times will I need to flip this coin in order for a heads to appear?

2 Hashing

Remember from class that a hash function is a mapping $h : \{0, \ldots, u-1\} \to \{0, \ldots, m-1\}$. Intuitively, you should think of u as the size of the universe, and m as the size of the short phrase we are assigning to each element of the universe. Most of the time, $u >> m$. Using the birthday paradox as an example, we would have $u = 7,000,000,000$ and $m = 365$ where each of the 7 billion+ people (with some ID numbering system) would have a birthday as one of the 365 days of the year.

Even though our hash function is defined for all 7 billion+ people in the world, we are only going to examine a subset of the population at once. Notation-wise, we typically use n to denote the number of items that we are going to hash. For example, if I wanted to figure out how many people taking CS 124 this spring have the same birthday, n would be equal to 220. We are interested in how many of these 220 people (not the entire 7 billion) share the same birthday.

Hashing-based data structures are useful since they ideally allow for constant time operations (lookup, adding, and deletion), although collisions can change that if, for example, m is too small or you use a poorly chosen hash function. Nevertheless, hash tables can help us maintain a set of items and quickly answer whether a given item is in our set of items.

In this class, we will always be working with the assumption that h is a **Perfectly Random Hash Function**. This means that for any x in $\{0, 1, 2 \ldots u-1\}$, the probability that $h(x) = y$ for any $y \in \{0, 1, 2 \ldots m-1\}$ is $\frac{1}{m}$. In other words, $h(x)$ is equally likely to be any of the m possible values. **Warning:** This does not mean that the evaluation of h is random, but rather the way we choose h out of all possible functions from $\{0, 1, 2 \ldots u-1\}$ to $\{0, 1, 2 \ldots m-1\}$ is random.

Exercise 2. Suppose I hash n items into m buckets with a perfectly random hash function.

(a) What's the expected number of buckets that have exactly one item?

(b) What's the expected number of buckets that have exactly 2 items?

(c) What's the expected number of buckets that have strictly more than 2 items?

There are several ways to deal with collisions while hashing, such as:

- Chaining: Each bucket holds a set of all items that are hashed to it. Simply add x to this set.
- Linear probing: If $f(x)$ already has an item, try $f(x)+1, f(x)+2$, etc. until you find an empty location (all taken mod m).
- Double hashing: Use two hash functions: $f(i,x) = f_1(x) + if_2(x)$. If $f(0,x)$ is taken, try $f(1,x), f(2,x), etc.$ until you find an empty location. This generalizes linear probing.
- Cuckoo hashing: Again, use two hash functions and place x in either $f_1(x)$ or $f_2(x)$. If there's a collision with object y, push y out to it's other location and keep repeating until there are no more collisions.

2.1 Bloom Filters

A Bloom Filter is a probabilistic data structure, used for set membership problems, that are more space efficient than conventional hashing schemes. There are m bits and k hash functions $f_1, \ldots, f_k$. When adding an element x to the set, set bits $f_1(x), \ldots, f_k(x)$ to 1. To check if x is already in the set, check if the corresponding bits are set to 1. Typically, the buckets are split up into k tables, with each hash function "addressing" a single table.

Tradeoff: With bloom filters, we trade away *correctness* for *space*. We know that when asked: "Is x in the bloom filter?", it is possible for x to be not in the bloom filter yet we say that it is. If we want to support such queries for u values of x with 100% accuracy, we would need u bits of memory. However, with bloom filters, we only need m bits of memory, where again $u >> m$.

Exercise 3. Can you delete a single element from a Bloom filter?

Bloom filters are probabilistic structures since it's possible to get false positives, but never false negatives. That is, querying for membership of y may return true if y hasn't been added to the set but will never return false if it has.

Exercise 4. What's the probability of getting a false positive in a bloom filter with n elements inserted already?

2.2 Fingerprinting

- Hash each set of $|P|$ consecutive characters into a 16 bit value (can be other sizes) by taking mods.
- Taking mods might take a long time if you just do it naively. Instead, let the leftmost digit be a of old number N, and new inserted rightmost digit be b of new number N'. Update by:

$$N' = (10(N - 10^{|P-1|} \cdot a) + b) \mod p$$

- Randomly picking primes: when we use randomized algorithms, we look at expected values instead of worse case scenarios. We want to bound the probability of something bad happening.
- Pick multiple primes to make the probability of a false positive really small.

Exercise 5. In class, we talked about the problem of pattern matching, trying to find a substring within longer string and how to solve that using the fingerprinting technique. In this problem, we will explore a similar application of fingerprinting, which is to *check* integer multiplication.

We are given three integers a, b and c and want to determine whether or not $a \cdot b = c$. Suppose that $0 \leq a, b < 10^{250,000}$ and $0 \leq c < 10^{500,000}$ so actually performing the multiplication would not be feasible. Suppose that we are given a, b and c as strings (and thus do not have to worry about integer overflow on our machines).

(a) If someone told you to check whether $23898239 \cdot 19392981 = 83431298313$ is true. How can you tell the answer is *false* immediately?

(b) Generalize your strategy to come up with an algorithm that tests whether $a \cdot b = c$. Be sure your algorithm is *randomized* so that an it works well on average for any a, b, c.

(c) Using the Prime Number Theorem, which says that there are $\Theta(n/\ln n)$ primes less than n, bound the failure probability of your algorithm.

(d) Describe in more detail how you would implement your algorithm from (b). Be sure to make it efficient.

3 Primality Testing

Primality testing is important because often times we will need a prime number p. We may need a p with hundreds of digits, so it's not feasible to check whether p is prime by checking all divisibility from 1 to $\sqrt{p}$. Therefore, we have the following algorithms that help us find primes a lot faster, but we trade away some accuracy.

3.1 Fermat Testing

Fermat's Little Theorem says that given prime p and $1 \leq a < p$, $a^{p-1} = 1 \mod p$. We can modify this to turn it into a primality test: pick a, and calculate $a^{n-1} \mod n$ for some randomly chosen $n > a$. A number n that passes this test is called a-pseudoprime. Note that we can calculate a^p efficiently using repeated squaring.

However, this doesn't always work—for instance, $341 = 11 \cdot 31$, but $2^{340} = 1 \mod 341$. But can't we just try with a different choice of a? Most of the time, we can. However, if n is a Carmichael number, n will be a-pseudoprime for *any* choice of a, and there are infinitely many of them. This is bad because if we used fermat testing as our primality testing algorithm, there would be certain inputs (i.e. the Carmichael numbers) that we would have a 0% success rate on.

3.2 Rabin-Miller Testing

Instead, we use the Rabin-Miller primality test. Given a prime candidate n, let u be such that $n-1 = 2^t u$. For some a, calculate a^u and its subsequent squares. If at any time we have $a^{2^{i-1}u} \neq \pm 1 \mod n$ and $a^{2^i u} = 1 \mod n$, we have a nontrivial square root of 1 modulo n and n must be composite. In such a situation, we call a a witness to the compositeness of n.

Proof. Suppose we have $x^2 = 1 \mod p$ but $x = b \mod p$ where $b \neq -1, 1$. Since we have $x^2 - 1 = k \cdot p$ for some k, this gives us $(x+1)(x-1) = kp$. Suppose p is prime. Then, we must have that $x + 1 = 0 \mod p$ or $x - 1 = 0 \mod p$ because the factor of p cannot be split among the $x + 1$ and $x - 1$. But this contradicts our assumption that $b \neq 1, -1$ and thus p cannot be prime. □

How good is the Rabin-Miller primality test? If n is composite, a randomly selected a will be a witness with probability at least $\frac{3}{4}$, which means that by checking very few a's, we can determine with high probability that any number n is prime.

Exercise 6. Consider the number 1105 (Assume you didn't know the divisibility rule for 5).

(a) Does 1105 pass Fermat's test?

(b) Does 1105 pass the Rabin-Miller test?