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? How many heads should you expect if you flip the coin 12 times?

(b) 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 $h : \{1, \ldots, U\} \rightarrow \{1, \ldots, m\}$. 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 320. We are interested in how many of these 320 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.

Exercise 2. Suppose I hash $n$ items into $m$ buckets with a perfectly random hash function (meaning that for any $x$, $h(x)$ is equally likely to be any of the $m$ possible values).

(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.

### 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 Universal Hash Families

- (From lecture notes) We say a family $\mathcal{H}$ of hash function mapping $\{1, \ldots, U\}$ into $\{1, \ldots, m\}$ is universal if for all $1 \leq x < y \leq U$,

$$\Pr_{h \in \mathcal{H}} (h(x) = h(y)) \leq \frac{1}{m},$$

where $h$ is chosen uniformly at random from $\mathcal{H}$.

**Exercise 5.** For some prime $p$, consider hashing $n$-digit strings consisting of letters from $\{0, 1, \ldots, p - 1\}$ (so here, $U = 2^n$) into the buckets $\{0, 1, \ldots, p - 1\}$ (so $m = p$) as follows: randomly generate the $n$-tuple $(c_1, c_2, \ldots, c_n)$ by selecting each component randomly from $\{0, 1, \ldots, p - 1\}$. Then for any key $\vec{x} = (x_1, x_2, \ldots, x_n)$, hash according to the function $h(\vec{x}) = \sum_{i=1}^{n} c_i x_i \mod p$. Prove that this process forms a universal hash family.
3 Skip Lists

(From lecture notes) The predecessor problem is to maintain a set of keys $k_1, k_2, \ldots$ subject to the following operations:

- **insert($k$)**: insert a new item with key $k$ into the database
- **delete($x$)**: delete item $x$ from the database (assuming $x$ is a pointer to the item)
- **pred($k$)**: return the item $k'$ in the database with $k'$ as large as possible such that $k' \leq k$ (note that $k' = k$ if it exists in the database)

Balanced binary search trees achieve a worst-case runtime of $O(\log n)$ for each of these three operations, but are generally relatively complicated to implement. In lecture, we introduced the skip list, a very simple randomized data structure which achieves an expected runtime of $O(\log n)$ for each of these operations.

**Exercise 6.** In class we showed an upper bound for the expected runtime of query($i$) on $n$ elements using $p = \frac{1}{2}$ for the probability of promoting elements to higher levels. What is a bound for the expected runtime using general $p$, and how can we choose $p$ to optimize this bound?