We use the Chernoff bound for the Poisson distribution (Theorem 5.4) to bound this probability, writing the bound as

$$\Pr(X \geq x) \leq e^{x-m-x\ln(x/m)}.$$

For $x = m + \sqrt{2m\ln m}$, we use that $\ln(1+z) \geq z - z^2/2$ for $z \geq 0$ to show

$$\begin{aligned}\Pr\left(X > m + \sqrt{2m\ln m}\right) &\leq e^{\sqrt{2m\ln m} - \left(m+\sqrt{2m\ln m}\right)\ln\left(1+\sqrt{2\ln m/m}\right)} \\ &\leq e^{\sqrt{2m\ln m} - \left(m+\sqrt{2m\ln m}\right)\left(\sqrt{2\ln m/m} - \ln m/m\right)} \\ &= e^{-\ln m + \sqrt{2m\ln m}(\ln m/m)} = o(1).\end{aligned}$$

A similar argument holds if $x < m$, so $\Pr\left(|X-m| > \sqrt{2m\ln m}\right) = o(1)$.

We now show the second fact, that

$$\left|\Pr\left(\mathcal{E} \mid |X-m| \leq \sqrt{2m\ln m}\right) - \Pr(\mathcal{E} \mid X = m)\right| = o(1).$$

Note that $\Pr(\mathcal{E} \mid X = k)$ is increasing in k, since this probability corresponds to the probability that all bins are nonempty when k balls are thrown independently and uniformly at random. The more balls that are thrown, the more likely all bins are nonempty. It follows that

$$\begin{aligned}\Pr\left(\mathcal{E} \mid X = m - \sqrt{2m\ln m}\right) &\leq \Pr\left(\mathcal{E} \mid |X-m| \leq \sqrt{2m\ln m}\right) \\ &\leq \Pr\left(\mathcal{E} \mid X = m + \sqrt{2m\ln m}\right).\end{aligned}$$

Hence we have the bound

$$\begin{aligned}&\left|\Pr\left(\mathcal{E} \mid |X-m| \leq \sqrt{2m\ln m}\right) - \Pr(\mathcal{E} \mid X = m)\right| \\ &\qquad \leq \Pr\left(\mathcal{E} \mid X = m + \sqrt{2m\ln m}\right) - \Pr\left(\mathcal{E} \mid X = m - \sqrt{2m\ln m}\right),\end{aligned}$$

and we show the right-hand side is $o(1)$. This is the difference between the probability that all bins receive at least one ball when $m - \sqrt{2m\ln m}$ balls are thrown and when $m + \sqrt{2m\ln m}$ balls are thrown. This difference is equivalent to the probability of the following experiment: we throw $m - \sqrt{2m\ln m}$ balls and there is still at least one empty bin, but after throwing an additional $2\sqrt{2m\ln m}$ balls, all bins are nonempty. In order for this to happen, there must be at least one empty bin after $m - \sqrt{2m\ln m}$ balls; the probability that one of the next $2\sqrt{2m\ln m}$ balls covers this bin is at most $\left(2\sqrt{2m\ln m}\right)/n = o(1)$ by the union bound. Hence this difference is $o(1)$ as well. ■

5.5. Application: Hashing

5.5.1. *Chain Hashing*

The balls-and-bins-model is also useful for modeling *hashing*. For example, consider the application of a password checker, which prevents people from using common, easily cracked passwords by keeping a dictionary of unacceptable passwords. When a user tries to set up a password, the application would like to check if the requested password is part of the unacceptable set. One possible approach for a password checker would be to store the unacceptable passwords alphabetically and do a binary search on

the dictionary to check if a proposed password is unacceptable. A binary search would require $\Theta(\log m)$ time for m words.

Another possibility is to place the words into bins and then search the appropriate bin for the word. The words in a bin would be represented by a linked list. The placement of words into bins is accomplished by using a *hash function*. A hash function f from a universe U into a range $[0, n-1]$ can be thought of as a way of placing items from the universe into n bins. Here the universe U would consist of possible password strings. The collection of bins is called a *hash table*. This approach to hashing is called *chain hashing*, since items that fall in the same bin are chained together in a linked list.

Using a hash table turns the dictionary problem into a balls-and-bins problem. If our dictionary of unacceptable passwords consists of m words and the range of the hash function is $[0, n-1]$, then we can model the distribution of words in bins with the same distribution as m balls placed randomly in n bins. We are making a rather strong assumption by presuming that our hash function maps words into bins in a fashion that appears random, so that the location of each word is independent and identically distributed. There is a great deal of theory behind designing hash functions that appear random, and we will not delve into that theory here. We simply model the problem by assuming that hash functions are random. In other words, we assume that (a) for each $x \in U$, the probability that $f(x) = j$ is $1/n$ (for $0 \leq j \leq n-1$) and that (b) the values of $f(x)$ for each x are independent of each other. Notice that this does not mean that every evaluation of $f(x)$ yields a different random answer! The value of $f(i)$ is fixed for all time; it is just equally likely to take on any value in the range.

Let us consider the search time when there are n bins and m words. To search for an item, we first hash it to find the bin that it lies in and then search sequentially through the linked list for it. If we search for a word that is not in our dictionary, the expected number of words in the bin the word hashes to is m/n. If we search for a word that is in our dictionary, the expected number of other words in that word's bin is $(m-1)/n$, so the expected number of words in the bin is $1+(m-1)/n$. If we choose $n = m$ bins for our hash table, then the expected number of words we must search through in a bin is constant. If the hashing takes constant time, then the total expected time for the search is constant.

The maximum time to search for a word, however, is proportional to the maximum number of words in a bin. We have shown that when $n = m$ this maximum load is $\Theta(\ln n / \ln \ln n)$ with probability close to 1, and hence with high probability this is the maximum search time in such a hash table. While this is still faster than the required time for standard binary search, it is much slower than the average, which can be a drawback for many applications.

Another drawback of chain hashing can be wasted space. If we use n bins for n items, several of the bins will be empty, potentially leading to wasted space. The space wasted can be traded off against the search time by making the average number of words per bin larger than 1.

5.5.2. *Hashing: Bit Strings*

If we want to save space instead of time, we can use hashing in another way. Again, we consider the problem of keeping a dictionary of unsuitable passwords. Assume that

a password is restricted to be eight ASCII characters, which requires 64 bits (8 bytes) to represent. Suppose we use a hash function to map each word into a 32-bit string. This string will serve as a short *fingerprint* for the word; just as a fingerprint is a succinct way of identifying people, the fingerprint string is a succinct way of identifying a word. We keep the fingerprints in a sorted list. To check if a proposed password is unacceptable, we calculate its fingerprint and look for it on the list, say by a binary search.[2] If the fingerprint is on the list, we declare the password unacceptable.

In this case, our password checker may not give the correct answer! It is possible for a user to input an acceptable password, only to have it rejected because its fingerprint matches the fingerprint of an unacceptable password. Hence there is some chance that hashing will yield a *false positive*: it may falsely declare a match when there is not an actual match. The problem is that – unlike fingerprints for human beings – our fingerprints do not uniquely identify the associated word. This is the only type of mistake this algorithm can make; it does *not* allow a password that is in the dictionary of unsuitable passwords. In the password application, allowing false positives means our algorithm is overly conservative, which is probably acceptable. Letting easily cracked passwords through, however, would probably not be acceptable.

To place the problem in a more general context, we describe it as an *approximate set membership* problem. Suppose we have a set $S = \{s_1, s_2, \ldots, s_m\}$ of m elements from a large universe U. We would like to represent the elements in such a way that we can quickly answer queries of the form "Is x an element of S?" We would also like the representation to take as little space as possible. In order to save space, we would be willing to allow occasional mistakes in the form of false positives. Here the unallowable passwords correspond to our set S.

How large should the range of the hash function used to create the fingerprints be? Specifically, if we are working with bits, how many bits should we use to create a fingerprint? Obviously, we want to choose the number of bits that gives an acceptable probability for a false positive match. The probability that an acceptable password has a fingerprint that is different from any specific unallowable password in S is $(1 - 1/2^b)$. It follows that if the set S has size m and if we use b bits for the fingerprint, then the probability of a false positive for an acceptable password is $1 - (1 - 1/2^b)^m \geq 1 - e^{-m/2^b}$. If we want this probability of a false positive to be less than a constant c, we need

$$e^{-m/2^b} \geq 1 - c,$$

which implies that

$$b \geq \log_2 \frac{m}{\ln(1/(1-c))}.$$

That is, we need $b = \Omega(\log_2 m)$ bits. On the other hand, if we use $b = 2\log_2 m$ bits, then the probability of a false positive falls to

$$1 - \left(1 - \frac{1}{m^2}\right)^m < \frac{1}{m}.$$

[2] In this case the fingerprints will be uniformly distributed over all 32-bit strings. There are faster algorithms for searching over sets of numbers with this distribution, just as Bucket sort allows faster sorting than standard comparison-based sorting when the elements to be sorted are from a uniform distribution, but we will not concern ourselves with this point here.

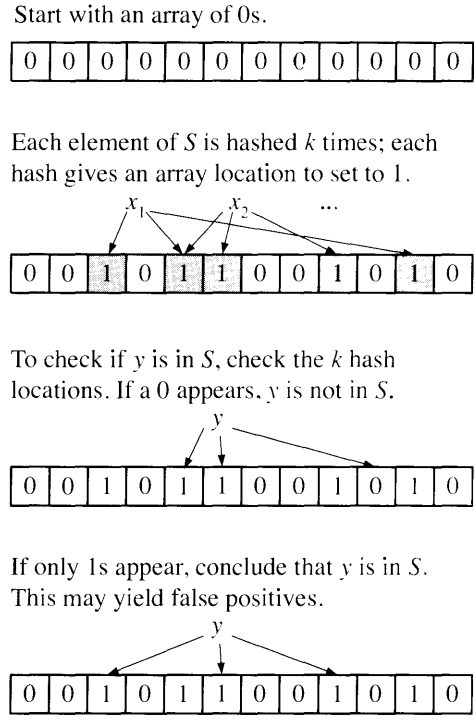

Figure 5.1: Example of how a Bloom filter functions.

In our example, if our dictionary has $2^{16} = 65{,}536$ words, then using 32 bits when hashing yields a false positive probability of just less than $1/65{,}536$.

5.5.3. *Bloom Filters*

We can generalize the hashing ideas of Sections 5.5.1 and 5.5.2 to achieve more interesting trade-offs between the space required and the false positive probability. The resulting data structure for the approximate set membership problem is called a *Bloom filter*.

A Bloom filter consists of an array of n bits, $A[0]$ to $A[n-1]$, initially all set to 0. A Bloom filter uses k independent random hash functions $h_1, \ldots, h_k$ with range $\{0, \ldots, n-1\}$. We make the usual assumption for analysis that these hash functions map each element in the universe to a random number uniformly over the range $\{0, \ldots, n-1\}$. Suppose that we use a Bloom filter to represent a set $S = \{s_1, s_2, \ldots, s_m\}$ of m elements from a large universe U. For each element $s \in S$, the bits $A[h_i(s)]$ are set to 1 for $1 \leq i \leq k$. A bit location can be set to 1 multiple times, but only the first change has an effect. To check if an element x is in S, we check whether all array locations $A[h_i(x)]$ for $1 \leq i \leq k$ are set to 1. If not, then clearly x is not a member of S, because if x were in S then all locations $A[h_i(x)]$ for $1 \leq i \leq k$ would be set to 1 by construction. If all $A[h_i(x)]$ are set to 1, we assume that x is in S, although we could be wrong. We would be wrong if all of the positions $A[h_i(x)]$ were set to 1 by elements of S even though x is not in the set. Hence Bloom filters may yield false positives. Figure 5.1 shows an example.

The probability of a false positive for an element not in the set – the *false positive probability* – can be calculated in a straightforward fashion, given our assumption that the hash functions are random. After all the elements of S are hashed into the Bloom filter, the probability that a specific bit is still 0 is

$$\left(1 - \frac{1}{n}\right)^{km} \approx e^{-km/n}.$$

We let $p = e^{-km/n}$. To simplify the analysis, let us temporarily assume that a fraction p of the entries are still 0 after all of the elements of S are hashed into the Bloom filter.

The probability of a false positive is then

$$\left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)^k \approx (1 - e^{-km/n})^k = (1 - p)^k.$$

We let $f = (1 - e^{-km/n})^k = (1 - p)^k$. From now on, for convenience we use the asymptotic approximations p and f to represent (respectively) the probability that a bit in the Bloom filter is 0 and the probability of a false positive.

Suppose that we are given m and n and wish to optimize the number of hash functions k in order to minimize the false positive probability f. There are two competing forces: using more hash functions gives us more chances to find a 0-bit for an element that is not a member of S, but using fewer hash functions increases the fraction of 0-bits in the array. The optimal number of hash functions that minimizes f as a function of k is easily found taking the derivative. Let $g = k \ln(1 - e^{-km/n})$, so that $f = e^g$ and minimizing the false positive probability f is equivalent to minimizing g with respect to k. We find

$$\frac{dg}{dk} = \ln(1 - e^{-km/n}) + \frac{km}{n} \frac{e^{-km/n}}{1 - e^{-km/n}}.$$

It is easy to check that the derivative is zero when $k = (\ln 2) \cdot (n/m)$ and that this point is a global minimum. In this case the false positive probability f is $(1/2)^k \approx (0.6185)^{n/m}$. The false positive probability falls exponentially in n/m, the number of bits used per item. In practice, of course, k must be an integer, so the best possible choice of k may lead to a slightly higher false positive rate.

A Bloom filter is like a hash table, but instead of storing set items we simply use one bit to keep track of whether or not an item hashed to that location. If $k = 1$, we have just one hash function and the Bloom filter is equivalent to a hashing-based fingerprint system, where the list of the fingerprints is stored in a 0–1 bit array. Thus Bloom filters can be seen as a generalization of the idea of hashing-based fingerprints. As we saw when using fingerprints, to get even a small constant probability of a false positive required $\Omega(\log m)$ fingerprint bits per item. In many practical applications, $\Omega(\log m)$ bits per item can be too many. Bloom filters allow a constant probability of a false positive while keeping n/m, the number of bits of storage required per item, constant. For many applications, the small space requirements make a constant probability of error acceptable. For example, in the password application, we may be willing to accept false positive rates of 1% or 2%.

Bloom filters are highly effective even if $n = cm$ for a small constant c, such as $c = 8$. In this case, when $k = 5$ or $k = 6$ the false positive probability is just over 0.02. This contrasts with the approach of hashing each element into $\Theta(\log m)$ bits. Bloom filters require significantly fewer bits while still achieving a very good false positive probability.

It is also interesting to frame the optimization another way. Consider f, the probability of a false positive, as a function of p. We find

$$\begin{aligned} f &= (1-p)^k \\ &= (1-p)^{(-\ln p)(m/n)} \\ &= (e^{-\ln(p)\ln(1-p)})^{m/n}. \end{aligned} \tag{5.8}$$

From the symmetry of this expression, it is easy to check that $p = 1/2$ minimizes the false positive probability f. Hence the optimal results are achieved when each bit of the Bloom filter is 0 with probability $1/2$. An optimized Bloom filter looks like a random bit string.

To conclude, we reconsider our assumption that the fraction of entries that are still 0 after all of the elements of S are hashed into the Bloom filter is p. Each bit in the array can be thought of as a bin, and hashing an item is like throwing a ball. The fraction of entries that are still 0 after all of the elements of S are hashed is therefore equivalent to the fraction of empty bins after mk balls are thrown into n bins. Let X be the number of such bins when mk balls are thrown. The expected fraction of such bins is

$$p' = \left(1 - \frac{1}{n}\right)^{km}.$$

The events of different bins being empty are not independent, but we can apply Corollary 5.9, along with the Chernoff bound of Eqn. (4.6), to obtain

$$\Pr(|X - np'| \geq \varepsilon n) \leq 2e\sqrt{n}e^{-n\varepsilon^2/3p'}.$$

Actually, Corollary 5.11 applies as well, since the number of 0-entries – which corresponds to the number of empty bins – is monotonically decreasing in the number of balls thrown. The bound tells us that the fraction of empty bins is close to p' (when n is reasonably large) and that p' is very close to p. Our assumption that the fraction of 0-entries in the Bloom filter is p is therefore quite accurate for predicting actual performance.

5.5.4. *Breaking Symmetry*

As our last application of hashing, we consider how hashing provides a simple way to break symmetry. Suppose that n users want to utilize a resource, such as time on a supercomputer. They must use the resource sequentially, one at a time. Of course, each user wants to be scheduled as early as possible. How can we decide a permutation of the users quickly and fairly?

If each user has an identifying name or number, hashing provides one possible solution. Hash each user's identifier into 2^b bits, and then take the permutation given by