Probability Review

Since we are going to do a probabilistic analysis of randomized algorithms, let us review the handful of notations from discrete probability that we are going to use.

Suppose that A and B are two *events*, that is, things that may or may not happen. Then we have the *union bound*

$$\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$$

where equality holds if and only if A and B are *mutually exclusive* (i.e. $A \wedge B = \emptyset$). If A and B are *independent* then we also have

$$\Pr[A \wedge B] = \Pr[A] \cdot \Pr[B]$$

Informally, a *random variable* is an outcome of a probabilistic experiment, which has various possible values with various probabilities. (Formally, it is a function from the sample space to the reals, though the formal definition does not help intuition very much.)

The *expectation* of a random variable X is

$$\mathbb{E}[X] = \sum_{x} \Pr[X = x] \cdot x$$

where v ranges over all possible values that the random variable can take.

Throughout our probabilistic analysis, we will primarily make use of the following three useful facts about random variables:

- *Linearity of expectation*: if X and Y are two random variables, then $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$, and if v is a constant, then $\mathbb{E}[vX] = v \cdot \mathbb{E}[X]$.
- *Product formula for independent random variables*: If X and Y are independent, then $\mathbb{E}[XY] = (\mathbb{E}[X]) \cdot (\mathbb{E}[Y])$

- *Markov's inequality*: If X is a random variable that is always ≥ 0, then, for every $t > 0$, we have
$$\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$$

Linearity of expectation and the product rule greatly simplify the computation of the expectation of random variables that come up in applications (which are often sums or products of simpler random variables). Markov's inequality is useful because once we compute the expectation of a random variable we are able to make high-probability statements about it. For example, if we know that X is a non-negative random variable of expectation 100, we can say that there is a $\geq 90\%$ probability that $X \leq 1,000$.

Finally, the *variance* of a random variable X is

$$\mathbf{Var}[X] := \mathbb{E}[(X - \mathbb{E}[X])^2]$$

and the *standard deviation* of a random variable X is $\sigma_X = \sqrt{\mathbf{Var} X}$. The significance of this definition is that, if the variance is small, we can prove that X has, with high probability, values close to the expectation. That is, for every $t > 0$, we can use Markov's inequality to yield

$$\Pr[|X - \mathbb{E}[X]| \geq t] = \Pr[(X - \mathbb{E}[X])^2 \geq t^2] \leq \frac{\mathbf{Var}[X]}{t^2}$$

and, after the change of variable $t = c\sqrt{\mathbf{Var} X} = c\sigma_X$, we have

$$\Pr[|X - \mathbb{E}[X]| \geq c\sigma_X] \leq \frac{1}{c^2}$$

so, for example, there is at least a 99% probability that X is within 10 standard deviations of its expectation. The above inequality is called *Chebyshev's inequality.* (There are a dozen alternative spellings for Chebyshev's name; you may have encountered this inequality before, spelled differently.)

If one knows more about X, then it is possible to say a lot more about the probability that X deviates from the expectation. In several interesting cases, for example, the probability of deviating by c standard deviations is exponentially, rather than polynomially, small in c^2. The advantage of Chebyshev's inequality is that one does not need to know anything about X other than its expectation and its variance.

Two final things about variance:

- If $X_1, \ldots, X_n$ are pairwise independent random variables, then

$$\mathbf{Var}[X_1 + \cdots + X_n] = \mathbf{Var}[X_1] + \cdots + \mathbf{Var}[X_n]$$

- If X is a random variable whose only possible values are 0 and 1 (i.e. X is an *indicator*), then

$$\mathbb{E}[X] = \Pr[X = 1]$$

and

$$\mathbf{Var}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \leq \mathbb{E}[X^2] = \mathbb{E}[X] = \Pr[X = 1]$$

1 Sampling

Perhaps, the simplest and most fundamental algorithm for estimating a statistic is *random sampling.* To illustrate the algorithm, let us consider a toy example.

Suppose we have a population of 300 voters participating in an election with two parties A and B. Our goal is to determine the fraction of the population voting for A. A simple sub-linear time algorithm would be to sample t voters out of 300 uniformly at random, and compute the fraction of the sampled voters who vote for A.

Let p denote the fraction of the population voting for A. Suppose we sample t people, each one independently and uniformly at random from the population. Let X_i denote the indicator for whether the i^{th} sample, votes for A. Then, the algorithm's estimate for p is given by

$$\tilde{p} = \frac{1}{t}\sum_{i=1}^{t} X_i$$

It is easy to check that,

$$\begin{aligned}\mathbb{E}[X_i] &= \Pr[X_i = 1] \cdot 1 + \Pr[X_i = 0] \cdot 0 \\ &= p \cdot 1 + (1 - p) \cdot 0 = p,\end{aligned}$$

and therefore $\mathbb{E}[\tilde{p}] = \frac{1}{t}\sum_i \mathbb{E}[X_i] = p$. So the expected value of our algorithm's estimate $\tilde{p}$ is exactly equal to p (such an estimator is said to be *unbiased*).

The natural question to answer is, how many samples do we need in order to ensure that the estimate $\tilde{p}$ is, say, within 0.1 of the correct value p with probability 0.9? To answer this question, we will appeal to the following theorem.

Theorem 1 *(Chernoff/Hoeffding Bound) Suppose $X_1 \ldots, X_t$ are i.i.d random variables taking values in $\{0, 1\}$. Let $p = \mathbb{E}[X_i]$ and $\epsilon > 0$, then*

$$\Pr\left[\left|\frac{1}{t}\sum_{i=1}^{t} X_i - p\right| \geq \epsilon\right] \leq 2e^{-2\epsilon^2 t}$$

If you're curious, we provide a proof in Appendix A. However, feel free to skip the proof if you're not too interested.

In particular, this bound allows us to say that if we want to ensure

$$\Pr[\text{Estimate } \tilde{p} \text{ has an error } \geq \epsilon] \leq \delta,$$

then we need to set $t = \lceil \frac{1}{2\epsilon^2} \ln\left(\frac{2}{\delta}\right) \rceil$.

Notice that the number of samples needed to ensure that the estimate is within error ϵ with probability $1 - \delta$, is independent of the population size! In other words, to obtain a 0.1-approximate estimate with probability 0.9, we need to sample the same number of voters irrespective of whether the total population is 300 or 300 million. In this sense, the running time of random sampling algorithm is not just sub-linear in input, but independent of the input size!

2 Streaming

In this section and the next one we study memory-efficient algorithms that process a stream (i.e. a sequence) of data items and are able to, in real time, compute useful features of the data.

Consider a network router that is monitors its incoming internet traffic. The router encounters a stream of packets, and would like to estimate some statistics associated with the internet traffic.

Here, any algorithm we come up with has to face two severe restrictions:

- The space available at the router is too little to store all the different types of packets (e.g. IP addresses) seen in the day.
- The router sees the stream of traffic (the input) only once, i.e., the algorithm can read the stream only once.

An algorithm designed with these restrictions is known as a *streaming algorithm*. Specifically, in the streaming model, the input is a stream of symbols $x_1, \ldots, x_n$ from some domain, say $\{1, \ldots, R\}$ for some R (think of R as polynomially large in n, say $R = n^3$). Typically, we will use Σ to denote the domain of each symbol, and size of the domain is $|\Sigma|$. The goal is to compute some function $f(x_1, \ldots, x_n)$. However, the space available to the algorithm is *poly*$(\log n)$ and the algorithm reads the input stream exactly once in the order $x_1, \ldots, x_n$.

Abstracting away all the data except the labels, we can think of our input as being a stream

$$s_1, s_2, \ldots s_n$$

where each $s_i \in \Sigma$ is a label, and Σ is the set of all possible labels (e.g. all product identifiers, or all IP addresses). For a given stream $s_1, \ldots, s_n$, and for a label $a \in \Sigma$, we will define f_a to be the *frequency* of a in the stream, i.e. the number of times a appears in the stream.

In the next section, we will see one example of a streaming algorithm that:

> Finds the number of distinct labels in the stream (e.g. finding how many different IP addresses the traffic flowing through the router emanated from).

Most streaming problems have simple solutions in which we store the entire stream. We can store a list of all the distinct labels we have seen, and for each of them also store the number of occurrences. That is, the data structure would contain a pair (a, f_a) for every label a that appears at least once in the stream. Such a data structure can be maintained using $O(k \cdot (\log n + \log |\Sigma|)) \leq O(n \cdot (\log n + \log |\Sigma|))$ bits of memory. Alternatively, one could have an array of size $|\Sigma|$ storing f_a for every label a, using $O(|\Sigma| \cdot \log n)$ bits of memory. If the labels are IPv6 addresses, then $\Sigma = 2^{128}$, so the second solution is definitely not feasible; in a setup in which a stream might contain hundreds of millions of items or more, even a data structure of the first type is rather large.

A streaming algorithm makes one pass through the data, and typically uses only *poly*$(\log n)$ bits of memory; in practice, this is *much* smaller than the simple solutions described in the previous paragraph. For example, we will give a solution to the heavy hitter problem that uses $O((\log n)^2)$ bits of memory (in realistic implementations, the data structure requires only an array of size about 300-600, containing 32-bit integers) and solutions to the other problems that use $O(\log n)$ bits of memory (in realistic settings, the data structures are an array of size ranging from a few dozens to a few thousands 32-bit integers).

3 Counting Distinct Elements

Let us see a streaming algorithm that counts the number of distinct elements. For sake of concreteness, let us consider the following setup of the problem: you are given a sequence of words $w_1, \ldots, w_n$ (say a really large piece of literature) and the goal is to estimate the number of distinct words in the sequence.

A trivial solution would involve scanning the words $w_1, \ldots, w_n$ once, while remembering at each point in time, all the words seen. In general, this algorithm requires $\Omega(n)$-bits of memory. Now, we will see an algorithm that uses sub-linear space, in fact, *poly*$(\log n)$ space.

Distinct Elements Algorithm We are now ready to describe a streaming algorithm for counting the number of distinct words. Let Σ denote the set of all possible words. First, we will describe an idealized algorithm to illustrate the main idea.

```
1: Pick a hash function $h : \Sigma \to [0, 1]$
2: Compute the minimum hash value $\alpha = \min_i h(w_i)$ by going over the stream
3: Output $\frac{1}{\alpha}$
```

Algorithm 1: Counting Distinct Elements

The algorithm finds the minimum hash value of a word in the stream, and outputs its inverse. It is easy to check that the algorithm can be implemented with $O(\log n)$ bits of space (after suitably discretizing the hash function).

To describe the main idea behind the algorithm, let us first make a strong assumption about the hash function.

(Random Hash Assumption) *For each word $w \in \Sigma$, its hash value $h(w)$ is a uniformly random number in $[0, 1]$ independent of all other hash values*

Now, we present the intuition behind the algorithm. Suppose the stream $w_1, \dots, w_n$ contains k different words. Then the algorithm encounters k different hash values. If $r_1, \dots, r_k$ are these k-different hash values, then the algorithm will output $\frac{1}{\min(r_1,\dots,r_k)}$.

If $r_1, \dots r_k$ are k independently chosen random numbers in $[0, 1]$ then we expect these numbers to be uniformly distributed in $[0, 1]$ and the smallest of them to be around $\frac{1}{k}$. Therefore, we can expect that the reciprocal of the minimum is approximately k – the number of distinct words.

More formally, one can prove the validity of this intuition through the following lemma:

Lemma 2 *If there are k distinct elements in the stream then $\mathbb{E}[\min_i h(w_i)] = \frac{1}{k+1}$*

If you're curious, we provide a proof in Appendix B. However, feel free to skip the proof if you're not too interested.

Now, we provide a simple calculation showing that there is at least 60% probability that the algorithm achieves a constant-factor approximation. Suppose that the stream has k distinct labels, and call $r_1, \dots, r_k$ the k random numbers in $[0, 1]$ corresponding to the evaluation of h at the distinct labels of the stream. Then

$$\begin{aligned}
\Pr\left[\text{Algorithm's output} \le \frac{k}{2}\right] &= \Pr\left[\min\{r_1,\ldots,r_k\} \ge \frac{2}{k}\right] \\
&= \Pr\left[r_1 \ge \frac{2}{k} \wedge \ldots \wedge r_k \ge \frac{2}{k}\right] \\
&= \Pr\left[r_1 \ge \frac{2}{k}\right] \cdot \ldots \cdot \Pr\left[r_k \ge \frac{2}{k}\right] \\
&= \left(1 - \frac{2}{k}\right)^k \\
&\le e^{-2} \le .14
\end{aligned}$$

and

$$\begin{aligned}
\Pr\left[\text{Algorithm's output } \ge 4k\right] &= \Pr\left[\min\{r_1,\ldots,r_k\} \le \frac{1}{4k}\right] \\
&= \Pr\left[r_1 \le \frac{1}{4k} \vee \ldots \vee r_k \le \frac{1}{4k}\right] \\
&\le \sum_{i=1}^{k} \Pr\left[r_i \le \frac{1}{4k}\right] \\
&= \frac{1}{4}
\end{aligned}$$

so that

$$\Pr\left[\frac{k}{2} \le \text{Algorithm's output } \le 4k\right] \ge .61$$

There are various ways to improve the quality of the approximation and to increase the probability of success.

One of the simplest ways is to keep track not of the smallest hash value encountered so far, but the t *distinct labels with the smallest hash values.* This is a set of labels and values that can be kept in a data structure of size $O(t \log |\Sigma|)$, and processing an element from the stream takes time $O(\log t)$ if the labels are put in a priority queue. Then, if t_{sh} is the t-th smallest hash value in the stream, our estimate for the number of distinct values is $\frac{t}{t_{\text{sh}}}$.

The intuition is that, as before, if we have k distinct labels, their hashed values will be k random points in the interval $[0, 1]$, which we would expect to be uniformly spaced, so that the t-th smallest would have a value of about $\frac{t}{k}$. Thus its inverse, multiplied by t, should be an estimate of k.

Why would it help to work with the t-th smallest hash instead of the smallest? The intuition is that it takes only one outlier to skew the minimum, but one needs to have t outliers to skew the t-th smallest, and the latter is a more unlikely event.

4 Pseudorandom functions

The above analysis elucidates the central idea underlying our streaming algorithm. However, it crucially relies on the assumption that the algorithm has access to a hash function h that is a *uniformly random* function from $\Sigma \to [0,1]$. Therefore, there are a few roadblocks to implementing the algorithm as described.

First, in an actual implementation, the output of a hash function h can only have a finite range, say $\{1,\ldots,R\}$ for some positive integer R. By setting $h'(x) = h(x)/R$, we can obtain a hash function that takes a discrete set of values in $[0,1]$. The analysis of the streaming algorithm can be modified to take this discretization into account. Specifically, one can show that discretization introduces an additive error of at most ϵ if we use a sufficiently large range R (say $R > n/\epsilon$). Hence, a *uniformly random hash function* $h : \Sigma \to \{1,\ldots,R\}$ would suffice to implement the algorithm.

To be sure, let us understand how one could in principle generate a *uniformly random* hash function $h : \Sigma \to \{1,\ldots,R\}$. Let us assume that the algorithm has access to uniformly random bits, as many as it needs. For each $x \in \Sigma$, we can pick a random number $\gamma \in \{1,\ldots,R\}$ and set $h(x) = \gamma$, and construct a look-up table for the function h. This would constitute a *uniformly random function* h.

However, a streaming algorithm would not be able to store such a lookup table that has $\Omega(|\Sigma|)$ entries. Furthermore, by virtue of being *uniformly random*, the function h cannot be represented in any way other than a lookup table. Consequently, a streaming algorithm cannot use a *uniformly random* hash function. Instead, we will have our streaming algorithm pick a hash function at random from a small family of functions.

Formally, a *hash family* $\mathcal{H}$ is just a family of functions each with domain Σ and range $\{1,\ldots,R\}$. To pick a random hash function from family $\mathcal{H} = \{h_1,\ldots,h_{|\mathcal{H}|}\}$, an algorithm needs to sample a random number $r \in \{1,\ldots,|\mathcal{H}|\}$ and then use the function $h := h_r$. To store the hash function h, it is enough to store the index r which takes $O(\log|\mathcal{H}|)$ bits.

For example, suppose $\mathcal{H}$ is the set of all possible functions from the domain to the range. Then, selecting a hash function h from $\mathcal{H}$ corresponds to sampling a *uniformly random* function. In this case, the function h is highly random, but the size of $\mathcal{H}$ and consequently the memory needed to store h is large. At the other extreme, suppose $\mathcal{H}$ consists of a fixed pair of functions $\{h_0, h_1\}$. A function h chosen from hash family $\mathcal{H}$ needs just one bit to memorize, but is very far from random. In fact, for each x,

$h(x)$ takes only one of at most two values.

Intuitively, we need a small hash family $\mathcal{H}$ such that a function h chosen randomly from $\mathcal{H}$ is *sufficiently random looking* for the streaming algorithm. In particular, we would like to be able to prove the correctness of the streaming algorithm when it uses a hash function from the family $\mathcal{H}$. It turns out that all we need from the hash family is the property of *pairwise independence.*

Definition 3 *(Pairwise independent hash family)*

A family of functions $\mathcal{H} = \{h_1, \ldots, h_M\}$ from a domain Σ to a range R, is said to be a pairwise-independent hash family if the following holds: If we pick a hash function h at random from $\mathcal{H}$, then on any pair of inputs $x, y \in \Sigma$, the behaviour of h exactly mimics that of a completely random function. Formally, for all $x \neq y \in \Sigma$ and $i, j \in R$,

$$\Pr_{h \in \mathcal{H}}[h(x) = i \wedge h(y) = j] = \frac{1}{|R|^2}$$

The above definition also implies that each hash value is uniformly distributed by itself, i.e., for every $a \in \Sigma$, and for every $r \in \{1, \ldots, R\}$

$$\Pr[h(a) = r] = \frac{1}{R}$$

However a hash function chosen from a pairwise-independent hash family only looks random on two inputs at a time. If we consider three different hashes $h(x), h(y)$ and $h(z)$ simultaneously, then the three values might not appear random at all.

Here is a simple and elegant example of a pairwise-independent hash family that is of small size.

> Fix a prime number p. For each $a, b \in \mathbb{Z}_p$, define $h_{a,b} : \mathbb{Z}_p \to \mathbb{Z}_p$ as follows:
>
> $$h_{a,b}(x) = a \cdot x + b \mod p \ ,$$
>
> and the hash family $\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p, b \in \mathbb{Z}_p\}$. $\mathcal{H}$ is a universal hash family such that any function in the family can be represented in $O(\log p)$ bits by storing two integers $a, b \in \mathbb{Z}_p$.

It is a good exercise to show that the above defined family is indeed pairwise independent. There are many other constructions of pairwise independent hash families (see Exercise 1.29 in DPV textbook).

In the next section, we include a proof of correctness of the streaming algorithm for distinct elements while using a pairwise independent hash family.

4.1 * Rigorous Analysis of the "t-th Smallest" Algorithm

Let us sketch a more rigorous analysis of the previously described algorithm: estimating the number of distinct labels in a stream by using the t-th smallest hash value. These calculations are a bit more complicated than the rest of the content of this lecture, so it is OK to skip them; we include it just for sake of completeness.

We will see that we can get an estimate that is, with high probability, between $k - \epsilon k$ and $k + \epsilon k$ for any $\epsilon > 0$, by choosing t to be of the order of $1/\epsilon^2$. For example, choosing $t = 30/\epsilon^2$ there is at least a 93% probability of getting an ϵ-approximation.

For concreteness, we will see that, with $t = 3,000$, we get, with probability at least 93%, an error that is at most 10%. As before, we let $r_1, \ldots, r_k$ be the hashes of the k distinct labels in the stream. We let t_{sh} be the t-th smallest of $r_1, \ldots, r_k$.

$$\begin{aligned}
\Pr[\text{Algorithm's output } \geq 1.1k] &= \Pr\left[t_{\text{sh}} \leq \frac{t}{1.1k}\right] \\
&= \Pr\left[\left(\#i : r_i \leq \frac{t}{1.1k}\right) \geq t\right]
\end{aligned}$$

Now let's study the number of is such that $r_i \leq t/(1.1k)$, and let's give it a name

$$N := \#i : r_i \leq \frac{t}{1.1k}$$

This is a random variable whose expectation is easy to compute. If we define indicator S_i to be 1 if $r_i \leq t/(1.1k)$ and 0 otherwise, then $N = \sum_i S_i$ and so

$$\mathbb{E}[N] = \sum_i \mathbb{E}[S_i] = \sum_i \Pr\left[r_i \leq \frac{t}{1.1k}\right] = k \cdot \frac{t}{1.1k} = \frac{t}{1.1}$$

The variance of N is also easy to compute.

$$\mathbf{Var}[N] = \sum_i \mathbf{Var}[S_i] \leq k \cdot \frac{t}{1.1k} \leq t$$

So N has an average of about $.91t$ and a standard deviation of less than $\sqrt{t}$, so by Chebyshev's inequality

$$\begin{aligned}
\Pr[\text{Algorithm's output } \geq 1.1k] &= \Pr[N \geq t] \\
&= \Pr[(N - \mathbb{E}\, N) \geq t - t/1.1] \\
&\leq \frac{\mathbf{Var}[N]}{(t/11)^2} \\
&= \frac{121}{t} \\
&= \frac{121}{3000} \leq 4.1\%
\end{aligned}$$

Similarly,

$$\begin{aligned}
\Pr[\text{Algorithm's output } \leq .9k] &= \Pr\left[t_{\text{sh}} \geq \frac{t}{.9k}\right] \\
&= \Pr\left[\left(\#i : r_i \leq \frac{t}{.9k}\right) \leq t\right]
\end{aligned}$$

and if we call

$$N := \#i : r_i \leq \frac{t}{.9k}$$

We see that

$$\begin{aligned}
\mathbb{E}[N] &= \frac{t}{.9} \\
\mathbf{Var}[N] &\leq t
\end{aligned}$$

and

$$\begin{aligned}
\Pr[\text{Algorithm's output } \leq .9k] &= \Pr[N \leq t] \\
&= \Pr\left[\mathbb{E}[N] - N \geq \frac{t}{.9} - t\right] \\
&\leq \frac{\mathbf{Var}[N]}{(t/9)^2} \\
&= \frac{81}{t} \\
&= \frac{81}{3000} = 2.7\%
\end{aligned}$$

So all together we have

$$\Pr[.9k \leq \text{Algorithm's output } \leq 1.1k] \geq 93\%$$

Notice that the above analysis of the t-th smallest algorithm does not require h to be random function $h : \Sigma \to \{1/N, \ldots, 1\}$—we just need the calculation of the expectation and variance of the number of labels in a certain set whose hash is in a certain range. For this, we just need, for every $a \neq b$, the values $h(a)$ and $h(b)$ to be independently distributed. Indeed, even if there was a small correlation between the distribution of $h(a)$ and $h(b)$, this could also be absorbed into the error calculations.

A Proof of Theorem 1

Proof: It is sufficient to prove one side of the bound, i.e.

$$\Pr\left[\frac{1}{t}\sum_{i=1}^{t} X_i - p \geq \epsilon\right] \leq e^{-2\epsilon^2 t}$$

because it applies to both sides of the mean (which implies the full two-sided bound).

Hence, we proceed by applying Markov's inequality:

$$\Pr\left[\frac{1}{t}\sum_{i=1}^{t} X_i - p \geq \epsilon\right] = \Pr\left[\sum_{i=1}^{t}(X_i - p) \geq t\epsilon\right] = \Pr\left[e^{s\sum_{i=1}^{t}(X_i-p)} \geq e^{st\epsilon}\right] \leq \frac{\mathbb{E}\left[e^{s\sum_{i=1}^{t}(X_i-p)}\right]}{e^{st\epsilon}}$$

where s is any arbitrary positive constant. Now, the crux of the proof is analyzing $\mathbb{E}\left[e^{s\sum_{i=1}^{t}(X_i-p)}\right]$. In particular, we use Hoeffding's lemma on each $\mathbb{E}[e^{s(X_i-p)}]$:

$$\mathbb{E}[e^{s(X_i-p)}] \leq e^{\frac{s^2}{8}}.$$

The proof for the general Hoeffding's lemma is out of scope for this class (since it uses properties of convex functions). Now, we can use the independence of the X_i's to bound $\mathbb{E}\left[e^{s\sum_{i=1}^{t}(X_i-p)}\right]$:

$$\mathbb{E}\left[e^{s\sum_{i=1}^{t}(X_i-p)}\right] = \mathbb{E}\left[\prod_{i=1}^{t} e^{s(X_i-p)}\right] = \prod_{i=1}^{t}\mathbb{E}\left[e^{s(X_i-p)}\right] \leq \prod_{i=1}^{t} e^{\frac{s^2}{8}} = e^{\frac{ts^2}{8}}$$

Hence, plugging this back into our original inequality, we have

$$\Pr\left[\frac{1}{t}\sum_{i=1}^{t} X_i - p \geq \epsilon\right] \leq \frac{\mathbb{E}\left[e^{s\sum_{i=1}^{t}(X_i-p)}\right]}{e^{st\epsilon}} \leq \frac{e^{\frac{ts^2}{8}}}{e^{st\epsilon}}$$

Finally, since $\epsilon > 0$, we can set $s = 4\epsilon$ to yield

$$\Pr\left[\frac{1}{t}\sum_{i=1}^{t} X_i - p \geq \epsilon\right] \leq e^{\frac{t}{8}\cdot(4\epsilon)^2-(4\epsilon)t\epsilon} = e^{-2\epsilon^2 t}$$

As mentioned at the beginning of this proof, this implies the two-sided bound:

$$\Pr\left[\left|\frac{1}{t}\sum_{i=1}^{t} X_i - p\right| \geq \epsilon\right] \leq 2e^{-2\epsilon^2 t}.$$

□

B Proof of Lemma 2

Proof: The lemma is equivalent to saying that if $r_1, \ldots, r_k$ are independently and uniformly distributed in $[0,1]$ then,

$$\mathbb{E}[\min(r_1, \ldots, r_k)] = \frac{1}{k+1}$$

We prove this property as follows:

$$\begin{aligned}\mathbb{E}[\min(r_1, \ldots, r_k)] &= \int_{r_1,\ldots,r_k} \min(r_1, \ldots, r_k)\, dr_1 dr_2 \ldots dr_k \\ &= \int_{r_1,\ldots,r_k} \left(\int_{r_{k+1}=0}^{1} \mathbf{1}\{r_{k+1} \leq \min(r_1, \ldots, r_k)\} dr_{k+1}\right) dr_1 dr_2 \ldots dr_k \\ &= \Pr_{r_1,\ldots,r_k,r_{k+1}}[r_{k+1} \leq \min(r_1, \ldots, r_k)]\end{aligned}$$

where $r_1, \ldots, r_k, r_{k+1}$ are uniformly random elements in $[0,1]$. But, $r_{k+1} \leq \min(r_1, \ldots, r_k)$ if and only if $r_{k+1} = \min(r_1, \ldots, r_{k+1})$. The claim follows by observing that $r_{k+1} = \min(r_1, \ldots, r_{k+1})$ with probability exactly $\frac{1}{k+1}$, since any of the $k+1$ elements $\{r_1, \ldots, r_{k+1}\}$ is equally likely to be the smallest element. □