Lecture 6: Basic Derandomization Techniques II

February 20, 2007

Based on scribe notes by Chun-Yun Hsiao and Vinod Vaikuntanathan. Lecture given by Dan Gutfreund.

1 Recap

In the previous lecture, we saw several derandomization techniques (enumeration, nonuniformity, nondeterminism) that are general in the sense that they apply to all of **BPP**, but are infeasible in the sense that they cannot be implemented by efficient deterministic algorithms. Today, we will see two derandomization techniques that can be implemented efficiently, but do not apply to all randomized algorithms.

2 The Method of Conditional Expectations

The general approach. Consider a randomized algorithm that uses m random bits. We can view all its sequences of coin tosses as corresponding to a binary tree of depth m. We know that most paths (from the root to the leaf) are "good," i.e., give the correct answer. A natural idea is to try and find such a path by walking down from the root and making "good" choices at each step. Equivalently, we try to find a good sequence of coin tosses "bit-by-bit". . The idea is simple: we find a good path bit-by-bit.

For $1 \le i \le m$ and $r_1, r_2, \ldots, r_i \in \{0,1\}$, define $P(r_1, r_2, \ldots, r_i)$ to be the fraction of continuations that are good sequences of coin tosses. More precisely, for $1 \le j \le m$, let R_j be a random variable over $\{0,1\}$ with equal probability,

$$\begin{aligned} P(r_1, r_2, \ldots, r_i) &\stackrel{\text{def}}{=} \Pr_{R_1, R_2, \ldots, R_m}[A(x; R_1, R_2, \ldots, R_m) \text{ is correct } | R_1 = r_1, R_2 = r_2, \ldots, R_i = r_i] \\ &= \mathop{\mathrm{E}}_{R_{i+1}}[P(r_1, r_2, \ldots, r_i, R_{i+1})]. \end{aligned}$$

By averaging, there exists an $r_{i+1} \in \{0,1\}$ such that $P(r_1, r_2, \ldots, r_i, r_{i+1}) \ge P(r_1, r_2, \ldots, r_i)$. So at $r_1, r_2, \ldots, r_i$, we simply pick r_{i+1} that maximizes $P(r_1, r_2, \ldots, r_i, r_{i+1})$. At the end we have $r_1, r_2, \ldots, r_m$, and

$$P(r_1, r_2, \ldots, r_m) \ge P(r_1, r_2, \ldots, r_{m-1}) \ge \cdots \ge P(r_1) \ge P(\Lambda) \ge 2/3$$

where $P(\Lambda)$ denotes the fraction of good paths from the root. Then $P(r_1, r_2, \ldots, r_m) = 1$, since it is either 1 or 0.

Note that to implement this method, we need to compute $P(r_1, r_2, \ldots, r_i)$ deterministically, and this may be infeasible. However, there are nontrivial algorithms where this method does work,

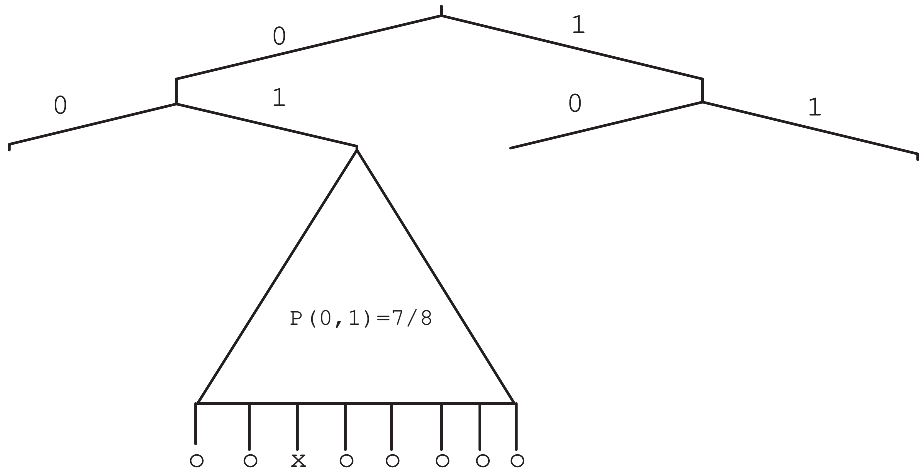

Figure 1: An example of $P(r_1, r_2)$, where "o" at the leaf denotes a good path.

often for *search* problems rather than decision problems. Below we see one such example, where it turns out to yield a natural "greedy algorithm".

Example. Recall the LARGE CUT problem: given a graph $G = (V, E)$, find a partition S, T (i.e., $S \cap T = \emptyset$, $S \cup T = V$) such that $|\mathrm{cut}(S, T)| \geq |E|/2$.[1]

We saw a simple randomized algorithm that finds a cut of (expected) size at least $|E|/2$, which we now phrase in a way suitable for derandomization.

Randomized LARGE CUT Algorithm: Flip $|V|$ coins $r_1, r_2, \ldots, r_{|V|}$, put vertex i in S if $r_i = 1$ and in T if $r_i = 0$.

To derandomize this algorithm using the Method of Conditional Expectations, define

$$e(r_1, r_2, \ldots, r_i) \stackrel{\mathrm{def}}{=} \mathop{\mathrm{E}}_{R_1, R_2, \ldots, R_{|V|}} \left[|\mathrm{cut}(S, T)| \middle| R_1 = r_1, R_2 = r_2, \ldots, R_i = r_i \right]$$

to be the expected cut size when the first i random bits are fixed to $r_1, r_2, \ldots, r_i$.

We know that when no random bits are fixed, $e[\Lambda] \geq |E|/2$ (because each edge is cut with probability $1/2$), and all we need to calculate is $e(r_1, r_2, \ldots, r_i)$ for $1 \leq i \leq n$. For this particular algorithm it turns out that this quantity is not hard to compute. Let $S_i \stackrel{\mathrm{def}}{=} \{j : j \leq i, r_j = 1\}$ (resp. $T_i \stackrel{\mathrm{def}}{=} \{j : j \leq i, r_j = 0\}$) be the set of vertices in S (resp. T) after we determine r_i, and $U_i \stackrel{\mathrm{def}}{=} \{i+1, i+2, \ldots, n\}$ be the "undecided" vertices that have not been put into S or T. Then

$$e(r_1, r_2, \ldots, r_i) = |\mathrm{cut}(S_i, T_i)| + 1/2 \left(|\mathrm{cut}(S_i, U_i)| + |\mathrm{cut}(T_i, U_i)| + |\mathrm{cut}(U_i, U_i)| \right) \tag{1}$$

[1] Recall $\mathrm{cut}(S, T) \stackrel{\mathrm{def}}{=} \{(s, t) : (s, t) \in E,\ s \in S,\ t \in T\}$.

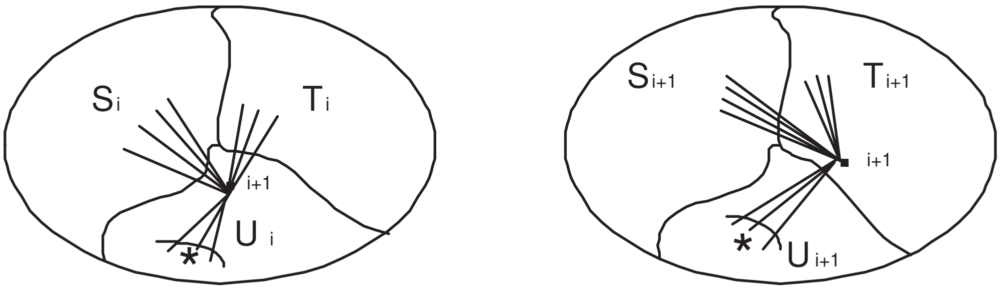

Figure 2: Marked edges are subtracted in $\text{cut}(U, U)$ but added back in $\text{cut}(T, U)$. The total change is the unmarked edges.

where $\text{cut}(U_i, U_i) = \{(u, v) : u, v \in U_i,\ (u, v) \in E\}$ is the set of (unordered) edges in the subgraph U_i. Now we can deterministically select a value for r_{i+1}, by computing and comparing $e(r_1, r_2, \ldots, r_i, 0)$ and $e(r_1, r_2, \ldots, r_i, 1)$.

In fact, the decision on R_{i+1} can be made even simpler than computing (1) in its entirety. Observe, in (1), that whether R_{i+1} equals 0 or 1 does not affect the *change* of the quantity in the parenthesis. That is,

$$(|\text{cut}(S_{i+1}, U_{i+1})| + |\text{cut}(T_{i+1}, U_{i+1})| + |\text{cut}(U_{i+1}, U_{i+1})|) - (|\text{cut}(S_i, U_i)| + |\text{cut}(T_i, U_i)| + |\text{cut}(U_i, U_i)|)$$

is the same regardless of whether $r_{i+1} = 0$ or $r_{i+1} = 1$. To see this, note that the only relevant edges are those having vertex $i+1$ as one endpoint. Of those edges, the ones whose other endpoint is also in U_i are removed from from the $\text{cut}(U, U)$ term but added to either the $\text{cut}(S, U)$ or $\text{cut}(T, U)$ term depending on whether $i + 1$ is put in S or T, respectively. (See Figure 2.) The edges with one endpoint being $i + 1$ and the other being in $S_i \cup T_i$ will be removed from the $|\text{cut}(S, U)| + |\text{cut}(T, U)|$ terms regardless of whether $i+1$ is put in S or T. Therefore, to maximize $e(r_1, r_2 \ldots, r_i, r_{i+1})$, it is enough to choose r_{i+1} that maximizes the $|\text{cut}(S, T)|$ term. This term increases by either $\text{cut}(\{i + 1\}, T_i)$ or $\text{cut}(\{i + 1\}, S_i)$ depending on whether we place vertex $i + 1$ in S or T, respectively. To summarize, we have

$$e(r_1, \ldots, r_i, 0) - e(r_1, \ldots, r_i, 1) = \text{cut}(\{i + 1\}, T_i) - \text{cut}(\{i + 1\}, S_i),$$

which gives rise to the following deterministic algorithm, which is guaranteed to always find a cut of size at least $|E|/2$:

Deterministic Large Cut **Algorithm:** On input $G = ([n], E)$,

1. Set $S = \emptyset$, $T = \emptyset$
2. For $i = 0, \ldots, n - 1$:
   (a) If $|\text{cut}(\{i + 1\}, S)| > |\text{cut}(\{i + 1\}, T)|$, set $T \leftarrow T \cup \{i + 1\}$,
   (b) Else set $S \leftarrow S \cup \{i + 1\}$.

Note that this is the natural 'greedy' algorithm for this problem. In other cases, the Method of Conditional Expectations yields algorithms that, while still arguably 'greedy', would have been much less easy to find directly. Thus, designing a randomized algorithm and then trying to derandomize it can be a useful paradigm for the design of deterministic algorithms even if the randomization does not provide gains in efficiency.

3 Pairwise Independence

As our first motivating example, we give another way of derandomizing the Large Cut algorithm discussed above. Recall the analysis of the randomized algorithm:

$$\mathrm{E}[|\mathrm{cut}(S)|] = \sum_{(i,j)\in E} \Pr[R_i \neq R_j] = |E|/2,$$

where $R_1, \ldots, R_n$ are the random bits of the algorithm. The key observation is that this analysis applies for any distribution on $(R_1, \ldots, R_n)$ satisfying $\Pr[R_i \neq R_j] = 1/2$ for each $i \neq j$. Thus, they do not need to be completely independent random variables; it suffices for them to be *pairwise independent*. That is, each R_i is an unbiased random bit, and for each $i \neq j$, R_i is independent from R_j.

This leads to the question: Can we generate N pairwise independent bits using less than N truly random bits ?

Proposition 1 *Let $B_1, \ldots, B_k$ be k independent unbiased random bits. For each nonempty $S \subseteq [k]$, let R_S be the random variable $\oplus_{i\in S} b_i$. Then the R_S's are $2^k - 1$ pairwise independent unbiased random bits.*

Proof: It is evident that each R_S is unbiased. For pairwise independence, consider any two nonempty sets $S \neq T \subseteq [k]$. Then:

$$\begin{aligned} R_S &= R_{S\cap T} \oplus R_{S\setminus T} \\ R_T &= R_{S\cap T} \oplus R_{T\setminus S}. \end{aligned}$$

Note that $R_{S\cap T}$, $R_{S\setminus T}$ and $R_{T\setminus S}$ are independent as they depend on disjoint subsets of the B_i's, and at least two of these subsets are nonempty (because $S \neq T$). This implies that (R_S, R_T) takes each value in $\{0,1\}^2$ with probability 1/4. ∎

Note that this gives us a way to generate N pairwise independent bits from $\lceil \log(N+1) \rceil$ independent random bits. Thus, we can reduce the randomness required by the Large Cut algorithm to logarithmic, and then we can obtain a deterministic algorithm by enumeration.

Derandomized Large Cut **Algorithm:** For *all* sequences of bits $b_1, b_2, \ldots, b_{\lceil \log(n+1) \rceil}$, run the randomized Large Cut algorithm using coin tosses $\{r_S = \oplus_{i\in S} b_i\}_{S\neq\emptyset}$ and choose the largest cut thus obtained.

Since there are at most $2(n+1)$ sequences of b_i's, the derandomized algorithm still runs in poly(n) time. It is slower than the greedy algorithm obtained by the Method of Conditional Expectations, but it has the advantage of using only $O(\log n)$ workspace and being parallelizable.

4 Pairwise Independent Hash Functions

Some applications require pairwise independent random variables that take values from a larger range, e.g. we want $N = 2^n$ pairwise independent random variables, each of which is uniformly distributed in $\{0,1\}^m = [M]$. The naïve approach is to repeat the above algorithm for the individual bits m times. This uses $(\log M)(\log N)$ bits to start with, which is no longer logarithmic in N if M is nonconstant. It turns out we can do much better.

A sequences of N random variables each taking a value in $[M]$ can be viewed as a distribution on sequences in $[M]^N$. Another interpretation of such a sequence is as a mapping $f : [N] \to [M]$. The latter interpretation turns out to be more useful when discussing the computational complexity of the constructions.

Definition 2 (Pairwise Independent Hash Functions) *A family of functions $\mathcal{H} = \{h : [N] \to [M]\}$ is* pairwise independent *if the following two conditions hold:*

1. *$\forall x \in [N]$, the random variable $H(x)$ is uniformly distributed in $[M]$ when $H \overset{R}{\leftarrow} \mathcal{H}$.*
2. *$\forall x_1 \neq x_2 \in [N]$, the random variables $H(x_1)$ and $H(x_2)$ are independent when $H \overset{R}{\leftarrow} \mathcal{H}$, .*

Equivalently, we can combine the two conditions to say that

$$\forall x_1 \neq x_2 \in [N], \forall y_1, y_2 \in [M], \Pr_{H \overset{\mathrm{R}}{\leftarrow} \mathcal{H}} [H(x_1) = y_1 \wedge H(x_2) = y_2] = \frac{1}{M^2}.$$

Note that the probability above is over the random choice of a function from the family $\mathcal{H}$. This is why we talk about a family of functions rather than a single function. The description in terms of functions makes it natural to impose a strong efficiency requirement — we ask that given the description of h and $x \in [N]$, the value $h(x)$ can be computed in time $\mathrm{poly}(\log N, \log M)$. We call such a family of hash functions *explicit*.

Pairwise independent functions are a strengthening of)*universal hash functions*, which require only that $\Pr[H(x_1) = H(x_2)] = 1/M$ for all $x \neq y$.

Below we present another construction of a pairwise independent family.

Proposition 3 *Let $\mathbb{F}$ be a finite field. Define the family of functions $\mathcal{H} = \{\ h_{a,b} : \mathbb{F} \to \mathbb{F}\}$ where each $h_{a,b}(x) = ax + b$ for $a, b \in \mathbb{F}$. Then $\mathcal{H}$ is pairwise independent.*

Proof Sketch: Notice that the graph of the function $h_{a,b}(x)$ is the line with slope a and y-intercept b. Given $x_1 \neq x_2$ and y_1, y_2, there is exactly one line containing the points (x_1, y_1) and (x_2, y_2). Thus, the probability over a, b that $h_{a,b}(x_1) = y_1$ and $h_{a,b}(x_2) = y_2$ equals the reciprocal of the number of lines, namely $1/\|F\|^2$. □

This construction uses $2 \log |\mathbb{F}|$ random bits, since we have to choose a and b at random from $\mathbb{F}$ to get a function $h_{a,b} \overset{\mathrm{R}}{\leftarrow} \mathcal{H}$. Compare this to $|\mathbb{F}| \log |\mathbb{F}|$ bits required to choose $|\mathbb{F}|$ fully independent values from $\mathbb{F}$, and $(\log |\mathbb{F}|)^2$ bits for repeating the construction of Proposition 1 for each output bit.

Note evaluating the functions of Proposition 3 requires an description of the field $\mathbb{F}$ which enables us to perform addition and multiplication of field elements. Recall that there is a (unique) finite field $\mathrm{GF}(p^t)$ of size p^t for every prime p and $t \in \mathbb{N}$. It is known how to construct a description of such a field (i.e. an irreducible polynomial of degree t over $\mathrm{GF}(p) = \mathbb{Z}_p$) in time $\mathrm{poly}(p, t)$. This satisfies our definition of explicitness when the prime p (the *characteristic* of the field) is small, e.g. for $p = 2$. Thus, we have an explicit construction of pairwise independent hash functions $\mathcal{H}_{n,n} = \{h : \{0,1\}^n \to \{0,1\}^n\}$ for every n.

What if we want a family $\mathcal{H}_{n,m}$ of pairwise independent hash functions where the input length n and output length m are not equal? For $n < m$, we can take hash functions h from $\mathcal{H}_{m,m}$ and restrict their domain to $\{0,1\}^m$ by defining $h'(x) = h(x \circ 0^{m-n})$. In the case that $m < n$, we can take h from $\mathcal{H}_{n,n}$ and throw away $n - m$ bits of the output. That is, define $h'(x) = h(x)|_m$, where $h(x)|_m$ denotes the first m bits of $h(x)$.

In both cases, we use $2\max\{n, m\}$ random bits. This is the best possible when $m \geq n$. When $m < n$, it can be reduced to $m + \max\{n, m\}$ random bits (which is optimal) by usin $(ax)|_m + b$ where $b \in \{0,1\}^m$ instead of $(ax + b)|_m$. Summarizing:

Theorem 4 *For every $n, m \in \mathbb{N}$, there is an explicit family of pairwise independent functions $\mathcal{H}_{n,m} = \{h : \{0,1\}^n \to \{0,1\}^m\}$ where a random function from $\mathcal{H}_{n,m}$ can be selected using $\max\{m, n\} + m$ random bits.*

5 Hash Tables

The original motivating application for pairwise independent functions was for hash tables. Suppose we want to store a set $S \subseteq [N]$ of values and answer queries of the form "*Is $x \in S$?*" efficiently (or, more generally, acquire some piece of data associated with key x in case $x \in S$). A simple solution is to have a table T such that $T[x] = 1$ if and only if $x \in S$. But this requires N bits of storage, which is inefficient if $|S| \ll N$.

A better solution is to use hashing. Assume that we have a hash function from $h : [N] \to [M]$ for some M to be determined later. Let the table T be of size M. For each $x \in [N]$, we let $T[h(x)] = x$ if $x \in S$. So to test whether a given $y \in S$, we compute $h(y)$ and check if $T[h(y)] = y$. In order for this construction to work, we need h to be one-to-one on the set S. Suppose we choose a random function H from $[N]$ to $[M]$. Then, for any set S, the expected number of collisions is

$$\mathrm{E}[\#\text{collisions}] = \sum_{x \neq y \in S} \Pr_h[h(x) = h(y)] = \binom{|S|}{2} \frac{1}{M} < \varepsilon$$

for $M = |S|^2/\varepsilon$. Notice that the above analysis does not require h to be a random function. It suffices that h be pairwise independent. We can generate and store h using $O(\log N)$ random bits. The storage required for the table T is $O(M \log N) = O(|S|^2 \log N)$. The space complexity can be improved to $O(|S| \log N)$, which is nearly optimal for small S, by taking $M = O(|S|)$ and using additional hash functions to separate the collisions.

Often, when people analyze applications of hashing in computer science, they model the hash function as a truly random function. However, the domain of the hash function is often exponentially large, and thus it is infeasible to even write down a truly random hash function. Thus, it would preferable to show that some explicit family of hash function works for the application with similar performance. In many cases, it can be shown that pairwise independence (or k-wise independence, as discussed below) suffices.

6 Randomness-Efficient Error Reduction and Sampling

Suppose we have a BPP algorithm for a language L that has a constant error probability. We want to reduce the error to 2^{-k}. We have already seen that using $O(k)$ independent repetitions, we can reduce the error of a BPP algorithm to 2^{-k} (using a Chernoff Bound). If the algorithm originally used m random bits, then we need $O(km)$ random bits after error reduction. Here we will see how to reduce the number of random bits required for error reduction by doing only pairwise independent repetitions.

To analyze this, we will need an analogue of the Chernoff Bound that applies to sums of pairwise independent random variables. This follows from Chebychev's Inequality. For a random variable X with expectation μ, recall that its *variance* is defined to be $\mathrm{Var}[X] = \mathrm{E}[(X-\mu)^2] = \mathrm{E}[X^2] - \mu^2$.

Lemma 5 (Chebyshev's Inequality) *Let X be a random variable with expectation μ, then*

$$\Pr[|X-\mu| \geq \varepsilon] \leq \frac{\mathrm{Var}[X]}{\varepsilon^2}$$

Proof: Let $Y = (X-\mu)^2$. Then

$$\Pr[|X-\mu| \geq \varepsilon] = \Pr[(X-\mu)^2 \geq \varepsilon^2] \leq \frac{\mathrm{E}[(X-\mu)^2]}{\varepsilon^2} = \frac{\mathrm{Var}[X]}{\varepsilon^2}.$$

■

We now use this to show that sums of pairwise independent random variables are concentrated around their expectation.

Proposition 6 (Pairwise-Independent Tail Inequality) *Let $X_1, \ldots, X_t$ be pairwise independent random variables taking values in the interval $[0,1]$, let $X = (\sum_i X_i)/t$, and $\mu = \mathrm{E}[X]$. Then*

$$\Pr[|X-\mu| \geq \varepsilon] \leq \frac{1}{t\varepsilon^2}.$$

Proof: Let $\mu_i = \mathrm{E}[X_i]$. Then

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{E}[(X-\mu)^2] \\
&= \frac{1}{t^2}\,\mathrm{E}[(\sum_i (X_i-\mu_i))^2] \\
&= \frac{1}{t^2}\sum_{i,j} \mathrm{E}[(X_i-\mu_i)(X_j-\mu_j)]
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{t^2}\sum_i \mathrm{E}[(X_i-\mu_i)^2] \qquad \text{(by pairwise independence)}\\
&= \frac{1}{t^2}\sum_i \mathrm{Var}[X_i]\\
&\le \frac{1}{t}
\end{aligned}
$$

Now apply Chebychev's Inequality. ■

While this requires less independence than the Chernoff Bound, notice that error probability decreases only linearly with t.

Error Reduction. Proposition 6 tells us that if we use $t = O(2^k)$ pairwise independent repetitions, we can reduce the error probability of a **BPP** algorithm from 1/3 to 2^{-k}. If the original **BPP** algorithm uses m random bits, then we can do this by choosing $h : \{0,1\}^k \to \{0,1\}^m$ at random from a pairwise independent family, and running the algorithm using coin tosses $h(x)$ for all $x \in \{0,1\}^k$ This requires $O(k+m)$ random bits.

	Number of Repetitions	Number of Random Bits
Independent Repetitions	$O(k)$	$O(km)$
Pairwise Independent Repetitions	$O(2^k)$	$O(k+m)$

Note that we have saved substantially on the number of random bits, but paid a lot in the number of repetitions needed. To maintain a polynomial-time algorithm, we can only afford $k = O(\log n)$. This setting implies that if we have a BPP algorithm with a constant error that uses m random bits, we have another BPP algorithm that uses $O(m+\log n) = O(m)$ random bits and has an error of $1/\mathrm{poly}(n)$. That is, we can go from constant to inverse-polynomial error only paying a constant factor in randomness.

Sampling. Recall the Sampling problem: Given an oracle to a function $f : \{0,1\}^m \to [0,1]$, we want to approximate $\mu(f)$ to within an additive error of ε.

We saw that we can solve this problem with probability $1-\delta$ by outputting the average of f on a random sample of $t = O(\log(1/\delta)/\varepsilon^2)$ points in $\{0,1\}^m$, where the correctness follows from the Chernoff Bound. To reduce the number of truly random bits used, we can use a pairwise independent sample instead. Specifically, taking $t = 1/(\varepsilon^2\delta)$ pairwise independent points, we get an error probability of at most δ. To generate t pairwise independent samples of m bits each, we need $O(m+\log(1/\varepsilon)+\log(1/\delta))$ truly random bits.

	Number of Samples	Number of Random Bits
Truly Random Sample	$O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$	$O(\frac{m}{\epsilon^2}\log\frac{1}{\delta})$
Pairwise Independent Repetitions	$O(\frac{1}{\epsilon^2\delta})$	$O(m+\log\frac{1}{\epsilon}+\log\frac{1}{\delta})$

k-wise Independence Our definition and construction of pairwise independent functions generalizes naturally to k-wise independence for any k.

Definition 7 (k-Wise Independent Hash Functions) *For $k \in \mathbb{N}$, a family of functions $\mathcal{H} = \{h : [N] \to [M]\}$ is* k-wise independent *if for all distinct $x_1, x_2, \ldots, x_k \in [N]$, the random variables $H(x_1), \ldots, H(x_k)$ are independent and uniformly distributed in $[M]$ when $H \stackrel{R}{\leftarrow} \mathcal{H}$, .*

Proposition 8 *Let $\mathbb{F}$ be a finite field. Define the family of functions $\mathcal{H} = \{\ h_{a_0,a_1,\ldots,a_k} : \mathbb{F} \to \mathbb{F}\}$ where each $h_{a_0,a_1,\ldots,a_{k-1}}(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{k-1}x^{k-1}$ for $a, b \in \mathbb{F}$. Then $\mathcal{H}$ is k-wise independent.*

Proof: Similarly to Proposition 3, it suffices to prove that for all distinct $x_1, \ldots, x_k \in \mathbb{F}$ and all $y_1, \ldots, y_k \in \mathbb{F}$, there is exactly one polynomial h of degree at most $k - 1$ such that $h(x_i) = y_i$ for all i. To show that such a polynomial exists, we can use the LaGrange Interpolation formula:

$$h(x) = \sum_{i=1}^{k} y_i \cdot \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}.$$

To show uniqueness, suppose we have two polynomials h and h' of degree at most $k - 1$ such that $h(x_i) = h'(x_i)$ for $i = 1, \ldots, k$. Then $h - h'$ has k roots, and thus must be the zero polynomial. ∎

Corollary 9 *For every $n, m, k \in \mathbb{N}$, there is a family of k-wise independent functions $\mathcal{H} = \{h : \{0,1\}^n \to \{0,1\}^m\}$ such that choosing a random function from $\mathcal{H}$ takes $k \cdot \max\{n, m\}$ random bits, and evaluating a function from $\mathcal{H}$ takes time $\mathrm{poly}(n, m, k)$.*

k-wise independent hash functions have similar types of applications to pairwise independent hash functions. The increased independence is crucial in derandomizing some algorithms. k-wise independent random variables also satisfy a tail bound similar to Proposition 6, with the key improvement being that the error probability vanishes linearly in $t^{k/2}$ rather than t.