3

Basic Derandomization Techniques

In the previous section, we saw some striking examples of the power of randomness for the design of efficient algorithms:

- Polynomial Identity Testing in **co-RP**.
- $[\times(1+\varepsilon)]$-Approx #DNF in **prBPP**.
- Perfect Matching in **RNC**.
- Undirected S-T Connectivity in **RL**.
- Approximating MaxCut in probabilistic polynomial time.

This was of course only a small sample; there are entire texts on randomized algorithms. (See the notes and references for Section 2.)

In the rest of this survey, we will turn toward *derandomization* — trying to remove the randomness from these algorithms. We will achieve this for some of the specific algorithms we studied, and also consider the larger question of whether *all* efficient randomized algorithms can be derandomized. For example, does $\mathbf{BPP} = \mathbf{P}$? $\mathbf{RL} = \mathbf{L}$? $\mathbf{RNC} = \mathbf{NC}$?

In this section, we will introduce a variety of "basic" derandomization techniques. These will each be deficient in that they are either infeasible (e.g., cannot be carried out in polynomial time) or specialized (e.g., apply only in very specific circumstances). But it will be

useful to have these as tools before we proceed to study more sophisticated tools for derandomization (namely, the "pseudorandom objects" of Sections 4–7).

3.1 Enumeration

We are interested in quantifying how much savings randomization provides. One way of doing this is to find the smallest possible upper bound on the deterministic time complexity of languages in **BPP**. For example, we would like to know which of the following complexity classes contain **BPP**:

Definition 3.1 (Deterministic Time Classes). [1]

$$\begin{aligned}
\mathbf{DTIME}(t(n)) = \ & \{L : L \text{ can be decided deterministically} \\
& \quad \text{in time } O(t(n))\} \\
\mathbf{P} = \cup_c \mathbf{DTIME}(n^c) & \quad (\text{"polynomial time"}) \\
\tilde{\mathbf{P}} = \cup_c \mathbf{DTIME}(2^{(\log n)^c}) & \quad (\text{"quasipolynomial time"}) \\
\mathbf{SUBEXP} = \cap_\varepsilon \mathbf{DTIME}(2^{n^\varepsilon}) & \quad (\text{"subexponential time"}) \\
\mathbf{EXP} = \cup_c \mathbf{DTIME}(2^{n^c}) & \quad (\text{"exponential time"})
\end{aligned}$$

The "Time Hierarchy Theorem" of complexity theory implies that all of these classes are distinct, i.e., $\mathbf{P} \subsetneq \tilde{\mathbf{P}} \subsetneq \mathbf{SUBEXP} \subsetneq \mathbf{EXP}$. More generally, it says that $\mathbf{DTIME}(O(t(n)/\log t(n))) \subsetneq \mathbf{DTIME}(t(n))$ for any efficiently computable time bound t. (What is difficult in complexity theory is separating classes that involve different computational resources, like deterministic time versus nondeterministic time.)

Enumeration is a derandomization technique that enables us to deterministically simulate any randomized algorithm with an exponential slowdown.

Proposition 3.2. $\mathbf{BPP} \subset \mathbf{EXP}$.

---

[1] Often $\mathbf{DTIME}(\cdot)$ is written as $\mathbf{TIME}(\cdot)$, but we include the **D** to emphasize that it refers to deterministic rather than randomized algorithms. Also, in some contexts (e.g., cryptography), "exponential time" refers to $\mathbf{DTIME}(2^{O(n)})$ and "subexponential time" to be $\mathbf{DTIME}(2^{o(n)})$; we will use **E** to denote the former class when it arises in Section 7.

Proof. If L is in **BPP**, then there is a probabilistic polynomial-time algorithm A for L running in time $t(n)$ for some polynomial t. As in the proof of Theorem 2.30, we write $A(x;r)$ for As output on input $x \in \{0,1\}^n$ and coin tosses $r \in \{0,1\}^{m(n)}$, where we may assume $m(n) \leq t(n)$ without loss of generality. Then:

$$\Pr[A(x;r) \text{ accepts}] = \frac{1}{2^{m(n)}} \sum_{r \in \{0,1\}^{m(n)}} A(x;r)$$

We can compute the right-hand side of the above expression in deterministic time $2^{m(n)} \cdot t(n)$. □

We see that the enumeration method is *general* in that it applies to all **BPP** algorithms, but it is *infeasible* (taking exponential time). However, if the algorithm uses only a small number of random bits, it becomes feasible:

Proposition 3.3. If L has a probabilistic polynomial-time algorithm that runs in time $t(n)$ and uses $m(n)$ random bits, then $L \in \mathbf{DTIME}(t(n) \cdot 2^{m(n)})$. In particular, if $t(n) = \text{poly}(n)$ and $m(n) = O(\log n)$, then $L \in \mathbf{P}$.

Thus an approach to proving $\mathbf{BPP} = \mathbf{P}$ is to show that the number of random bits used by any **BPP** algorithm can be reduced to $O(\log n)$. We will explore this approach in Section 7. However, to date, Proposition 3.2 remains the best unconditional upper-bound we have on the deterministic time-complexity of **BPP**.

Open Problem 3.4. Is **BPP** "closer" to **P** or **EXP**? Is $\mathbf{BPP} \subset \tilde{\mathbf{P}}$? Is $\mathbf{BPP} \subset \mathbf{SUBEXP}$?

3.2 Nonconstructive/Nonuniform Derandomization

Next we look at a derandomization technique that can be implemented efficiently but requires some nonconstructive "advice" that depends on the input length.

Proposition 3.5. If $A(x;r)$ is a randomized algorithm for a language L that has error probability smaller than 2^{-n} on inputs x of length n, then for every n, there exists a fixed sequence of coin tosses r_n such that $A(x;r_n)$ is correct for all $x \in \{0,1\}^n$.

Proof. We use the Probabilistic Method. Consider R_n chosen uniformly at random from $\{0,1\}^{r(n)}$, where $r(n)$ is the number of coin tosses used by A on inputs of length n. Then

$$\begin{aligned}\Pr[\exists x \in \{0,1\}^n \text{ s.t. } A(x;R_n) \text{ incorrect on } x] &\leq \sum_x \Pr[A(x;R_n) \text{ incorrect on } x] \\ &< 2^n \cdot 2^{-n} = 1\end{aligned}$$

Thus, there exists a fixed value $R_n = r_n$ that yields a correct answer for all $x \in \{0,1\}^n$. □

The advantage of this method over enumeration is that once we have the fixed string r_n, computing $A(x;r_n)$ can be done in polynomial time. However, the proof that r_n exists is nonconstructive; it is not clear how to find it in less than exponential time.

Note that we know that we can reduce the error probability of any **BPP** (or **RP**, **RL**, **RNC**, etc.) algorithm to smaller than 2^{-n} by repetitions, so this proposition is always applicable. However, we begin by looking at some interesting special cases.

Example 3.6 (Perfect Matching). We apply the proposition to Algorithm 2.7. Let $G = (V,E)$ be a bipartite graph with m vertices on each side, and let $A^G(x_{1,1},\ldots,x_{m,m})$ be the matrix that has entries $A^G_{i,j} = x_{i,j}$ if $(i,j) \in E$, and $A^G_{i,j} = 0$ if $(i,j) \notin E$. Recall that the polynomial $\det(A^G(x))$ is nonzero if and only if G has a perfect matching. Let $S_m = \{0,1,2,\ldots,m2^{m^2}\}$. We argued that, by the Schwartz–Zippel Lemma, if we choose $\alpha \stackrel{R}{\leftarrow} S_m^{m^2}$ at random and evaluate $\det(A^G(\alpha))$, we can determine whether $\det(A^G(x))$ is zero with error probability at most $m/|S|$ which is smaller than 2^{-m^2}. Since a bipartite graph with m

vertices per side is specified by a string of length $n = m^2$, by Proposition 3.5 we know that for every m, there exists an $\alpha_m \in S_m^{m^2}$ such that $\det(A^G(\alpha_m)) \neq 0$ if and only if G has a perfect matching, for every bipartite graph G with m vertices on each side.

Open Problem 3.7. Can we find such an $\alpha_m \in \{0, \ldots, m2^{m^2}\}^{m^2}$ explicitly, i.e., *deterministically* and *efficiently*? An **NC** algorithm (i.e., parallel time polylog(m) with poly(m) processors) would put PERFECT MATCHING in deterministic **NC**, but even a subexponential-time algorithm would be interesting.

Example 3.8 (Universal Traversal Sequences). Let G be a connected d-regular undirected multigraph on n vertices. From Theorem 2.49, we know that a random walk of poly(n, d) steps from any start vertex will visit any other vertex with high probability. By increasing the length of the walk by a polynomial factor, we can ensure that *every* vertex is visited with probability greater than $1 - 2^{-nd\log n}$. By the same reasoning as in the previous example, we conclude that for every pair (n, d), there exists a *universal traversal sequence* $w \in \{1, 2, \ldots, d\}^{\text{poly}(n,d)}$ such that for every n-vertex, d-regular, connected G and every vertex s in G, if we start from s and follow w then we will visit the entire graph.

Open Problem 3.9. Can we construct such a universal traversal sequence explicitly (e.g., in polynomial time or even logarithmic space)?

There has been substantial progress toward resolving this question in the positive; see Section 4.4.

We now cast the nonconstructive derandomizations provided by Proposition 3.5 in the language of "nonuniform" complexity classes.

Definition 3.10. Let **C** be a class of languages, and $\mathbf{a} : \mathbb{N} \to \mathbb{N}$ be a function. Then **C**/**a** is the class of languages defined as follows: $L \in \mathbf{C}/\mathbf{a}$

if there exists $L' \in \mathbf{C}$, and $\alpha_1, \alpha_2, \ldots \in \{0,1\}^*$ with $|\alpha_n| \leq a(n)$, such that $x \in L \Leftrightarrow (x, \alpha_{|x|}) \in L'$. The αs are called the *advice* strings.

$\mathbf{P/poly}$ is the class $\bigcup_c \mathbf{P}/n^c$, i.e., polynomial time with polynomial advice.

A basic result in complexity theory is that $\mathbf{P/poly}$ is exactly the class of languages that can be decided by polynomial-sized Boolean circuits:

Fact 3.11. $L \in \mathbf{P/poly}$ iff there is a sequence of Boolean circuits $\{C_n\}_{n \in \mathbb{N}}$ and a polynomial p such that for all n

(1) $C_n : \{0,1\}^n \to \{0,1\}$ decides $L \cap \{0,1\}^n$
(2) $|C_n| \leq p(n)$.

We refer to $\mathbf{P/poly}$ as a "nonuniform" model of computation because it allows for different, unrelated "programs" for each input length (e.g., the circuits C_n, or the advice α_n), in contrast to classes like $\mathbf{P}$, $\mathbf{BPP}$, and $\mathbf{NP}$, that require a single "program" of constant size specifying how the computation should behave for inputs of arbitrary length. Although $\mathbf{P/poly}$ contains some undecidable problems,[2] people generally believe that $\mathbf{NP} \not\subset \mathbf{P/poly}$. Indeed, trying to prove lower bounds on circuit size is one of the main approaches to proving $\mathbf{P} \neq \mathbf{NP}$, since circuits seem much more concrete and combinatorial than Turing machines. However, this too has turned out to be quite difficult; the best circuit lower bound known for computing an explicit function is roughly $5n$.

Proposition 3.5 directly implies:

Corollary 3.12. $\mathbf{BPP} \subset \mathbf{P/poly}$.

A more general meta-theorem is that "nonuniformity is more powerful than randomness."

[2] Consider the unary version of the Halting Problem, which can be decided in constant time given advice $\alpha_n \in \{0,1\}$ that specifies whether the nth Turing machine halts or not.

3.3 Nondeterminism

Although physically unrealistic, nondeterminism has proven to be a very useful resource in the study of computational complexity (e.g., leading to the class **NP**). Thus it is natural to study how it compares in power to randomness. Intuitively, with nondeterminism we should be able to guess a "good" sequence of coin tosses for a randomized algorithm and then do the computation deterministically. This intuition does apply directly for randomized algorithms with 1-sided error:

Proposition 3.13. $\mathbf{RP} \subset \mathbf{NP}$.

Proof. Let $L \in \mathbf{RP}$ and A be a randomized algorithm that decides it. A polynomial-time verifiable witness that $x \in L$ is any sequence of coin tosses r such that $A(x;r) = \text{accept}$. □

However, for 2-sided error (**BPP**), containment in **NP** is not clear. Even if we guess a "good" random string (one that leads to a correct answer), it is not clear how we can verify it in polynomial time. Indeed, it is consistent with current knowledge that **BPP** equals **NEXP** (nondeterministic exponential time)! Nevertheless, there is a sense in which we can show that **BPP** is no more powerful than **NP**:

Theorem 3.14. If $\mathbf{P} = \mathbf{NP}$, then $\mathbf{P} = \mathbf{BPP}$.

Proof. For any language $L \in \mathbf{BPP}$, we will show how to express membership in L using two quantifiers. That is, for some polynomial-time predicate P,

$$x \in L \quad \Longleftrightarrow \quad \exists y \forall z\, P(x,y,z), \tag{3.1}$$

where we quantify over y and z of length $\text{poly}(|x|)$.

Assuming $\mathbf{P} = \mathbf{NP}$, we can replace $\forall z\, P(x,y,z)$ by a polynomial-time predicate $Q(x,y)$, because the language $\{(x,y) : \forall z\, P(x,y,z)\}$ is in $\mathbf{co\text{-}NP} = \mathbf{P}$. Then $L = \{x : \exists y\, Q(x,y)\} \in \mathbf{NP} = \mathbf{P}$.

To obtain the two-quantifier expression (3.1), consider a randomized algorithm A for L, and assume w.l.o.g. that its error probability

is smaller than 2^{-n} and that it uses $m = \text{poly}(n)$ coin tosses. Let $A_x \subset \{0,1\}^m$ be the set of coin tosses r for which $A(x;r) = 1$. We will show that if x is in L, there exist m "shifts" (or "translations") of A_x that cover all points in $\{0,1\}^m$. (Notice that this is a $\exists\forall$ statement.) Intuitively, this should be possible because A_x contains all but an exponentially small fraction of $\{0,1\}^m$. On the other hand if $x \notin L$, then no m shifts of A_x can cover all of $\{0,1\}^m$. Intuitively, this is because A_x is an exponentially small fraction of $\{0,1\}^m$.

Formally, by a "shift of A_x" we mean a set of the form $A_x \oplus s = \{r \oplus s : r \in A_x\}$ for some $s \in \{0,1\}^m$; note that $|A_x \oplus s| = |A_x|$. We will show

$$\begin{aligned}
x \in L \Rightarrow \exists s_1, s_2, \ldots, s_m \in \{0,1\}^m \forall r \in \{0,1\}^m \quad & r \in \bigcup_{i=1}^{m} (A_x \oplus s_i) \\
\Leftrightarrow \exists s_1, s_2, \ldots, s_m \in \{0,1\}^m \forall r \in \{0,1\}^m \quad & \bigvee_{i=1}^{m} (A(x; r \oplus s_i) = 1); \\
x \notin L \Rightarrow \forall s_1, s_2, \ldots, s_m \in \{0,1\}^m \exists r \in \{0,1\}^m \quad & r \notin \bigcup_{i=1}^{m} (A_x \oplus s_i) \\
\Leftrightarrow \forall s_1, s_2, \ldots, s_m \in \{0,1\}^m \exists r \in \{0,1\}^m \quad & \neg \bigvee_{i=1}^{m} (A(x; r \oplus s_i) = 1).
\end{aligned}$$

We prove both parts by the Probabilistic Method, starting with the second (which is simpler).

$x \notin L$ Let $s_1, \ldots, s_m$ be arbitrary, and choose $R \stackrel{\text{R}}{\leftarrow} \{0,1\}^m$ at random. Now A_x and hence each $A_x \oplus s_i$ contains less than a 2^{-n} fraction of $\{0,1\}^m$. So, by a union bound,

$$\begin{aligned}
\Pr[R \in \bigcup_i (A_x \oplus s_i)] &\leq \sum_i \Pr[R \in A_x \oplus s_i] \\
&< m \cdot 2^{-n} < 1,
\end{aligned}$$

for sufficiently large n. In particular, there exists an $r \in \{0,1\}^m$ such that $r \notin \bigcup_i (A_x \oplus s_i)$.

$x \in L$: Choose $S_1, S_2, \ldots, S_m \stackrel{R}{\leftarrow} \{0,1\}^m$. Then, for every fixed r, we have:

$$\begin{aligned}\Pr[r \notin \bigcup_i (A_x \oplus S_i)] &= \prod_i \Pr[r \notin A_x \oplus S_i] \\ &= \prod_i \Pr[S_i \notin A_x \oplus r] \\ &< (2^{-n})^m,\end{aligned}$$

since A_x and hence $A_x \oplus r$ contains more than a $1 - 2^{-n}$ fraction of $\{0,1\}^m$. By a union bound, we have:

$$\Pr[\exists r \quad r \notin \bigcup_i (A_x \oplus S_i)] < 2^m \cdot (2^{-n})^m \leq 1.$$

Thus, there exist $s_1, \ldots, s_m$ such that $\bigcup_i (A_x \oplus s_i)$ contains all points r in $\{0,1\}^m$. □

Readers familiar with complexity theory might notice that the above proof shows that **BPP** is contained in the second level of the *polynomial-time hierarchy* (**PH**). In general, the kth level of the **PH** contains all languages that satisfy a k-quantifier expression analogous to (3.1).

3.4 The Method of Conditional Expectations

In the previous sections, 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. In this section and the next one, we will see two derandomization techniques that sometimes can be implemented efficiently, but do not apply to all randomized algorithms.

3.4.1 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," that is, give a 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."

To make this precise, fix a randomized algorithm A and an input x, and let m be the number of random bits used by A on input x. For $1 \leq i \leq m$ 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, if $R_1, \ldots, R_m$ are uniform and independent random bits, then

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

(See Figure 3.1.)

By averaging, there exists an $r_{i+1} \in \{0,1\}$ such that $P(r_1, r_2, \ldots, r_i, r_{i+1}) \geq P(r_1, r_2, \ldots, r_i)$. So at node $(r_1, r_2, \ldots, r_i)$, 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

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

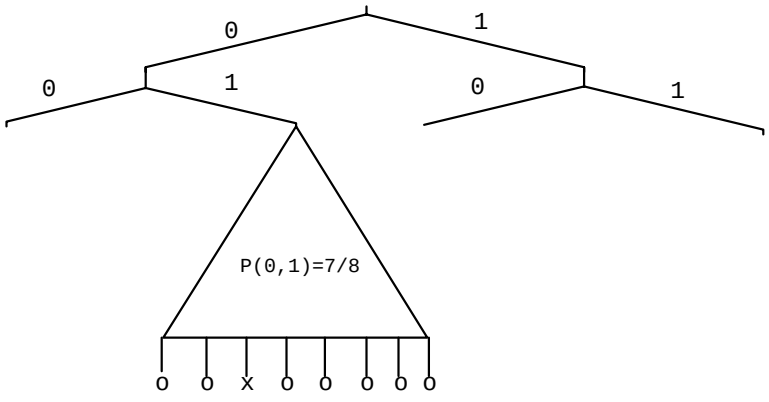

Fig. 3.1 An example of $P(r_1, r_2)$, where "o" at the leaf denotes a good path.

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, often for *search* problems rather than decision problems, and where we measure not a boolean outcome (e.g., whether A is correct as above) but some other measure of quality of the output. Below we see one such example, where it turns out to yield a natural "greedy algorithm."

3.4.2 Derandomized MAXCUT Approximation

Recall the MaxCutproblem:

Computational Problem 3.15 (Computational Problem 2.38, rephrased). MaxCut: Given a graph $G = (V, E)$, find a partition S, T of V (i.e., $S \cup T = V$, $S \cap T = \emptyset$) maximizing the size of the set $\text{cut}(S, T) = \{\{u, v\} \in E : u \in S, v \in T\}$.

We saw a simple randomized algorithm that finds a cut of (expected) size at least $|E|/2$ (not counting any self-loops, which can never be cut), which we now phrase in a way suitable for derandomization.

Algorithm 3.16 (randomized MaxCut, rephrased).
Input: A graph $G = ([N], E)$ (with no self-loops)

Flip N coins $r_1, r_2, \ldots, r_N$, put vertex i in S if $r_i = 0$ and in T if $r_i = 1$. Output (S, T).

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

$$e(r_1, r_2, \ldots, r_i) \stackrel{\text{def}}{=} \mathop{\mathrm{E}}_{R_1, R_2, \ldots, R_N} \Big[|\text{cut}(S, T)| \Big| R_1 = r_1, R_2 = r_2, \ldots, R_i = r_i \Big]$$

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

We know that when no random bits are fixed, $e[\Lambda] = |E|/2$ (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 the quantity is not hard to compute. Let $S_i \stackrel{\text{def}}{=} \{j : j \leq i, r_j = 0\}$ (resp. $T_i \stackrel{\text{def}}{=} \{j : j \leq i, r_j = 1\}$) be the set of vertices in S (resp. T) after we determine $r_1, \ldots, r_i$, and $U_i \stackrel{\text{def}}{=} \{i+1, i+2, \ldots, N\}$ be the "undecided" vertices that have not been put into S or T. Then

$$e(r_1, r_2, \ldots, r_i) = |\text{cut}(S_i, T_i)| + 1/2(|\text{cut}(U_i, [N])|). \tag{3.2}$$

Note that $\text{cut}(U_i, [N])$ is the set of *unordered* edges that have at least one endpoint in 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 (3.2) in its entirety, by observing that the set $\text{cut}(U_{i+1}, [N])$ is independent of the choice of r_{i+1}. 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

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

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

Algorithm 3.17 (deterministic MaxCut approximation).
Input: A graph $G = ([N], E)$ (with no self-loops)

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

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.

3.5 Pairwise Independence

3.5.1 An Example

As a motivating example for pairwise independence, we give another way of derandomizing the MaxCut approximation 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? The answer turns out to be *yes*, as illustrated by the following construction.

Construction 3.18 (pairwise independent bits). Let $B_1, \ldots, B_k$ be k independent unbiased random bits. For each nonempty $S \subset [k]$, let R_S be the random variable $\oplus_{i\in S} B_i$, where $\oplus$ denotes XOR.

Proposition 3.19. The $2^k - 1$ random variables R_S in Construction 3.18 are pairwise independent unbiased random bits.

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

$$R_S = R_{S\cap T} \oplus R_{S\setminus T}$$
$$R_T = R_{S\cap T} \oplus R_{T\setminus S}.$$

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_is, and at least two of these subsets are nonempty. 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 MaxCut algorithm to logarithmic, and then we can obtain a deterministic algorithm by enumeration.

Algorithm 3.20 (deterministic MaxCut algorithm II).
Input: A graph $G = ([N], E)$ (with no self-loops)

For *all* sequences of bits $b_1, b_2, \ldots, b_{\lceil \log(N+1) \rceil}$, run the randomized MaxCut 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_is, the derandomized algorithm still runs in polynomial time. It is slower than the algorithm we obtained by the Method of Conditional Expectations (Algorithm 3.17), but it has the advantage of using logarithmic workspace and being parallelizable. The derandomization can be sped up using almost pairwise independence (at the price of a slightly worse approximation factor); see Problem 3.4.

3.5.2 Pairwise Independent Hash Functions

Some applications require pairwise independent random variables that take values from a larger range, for example, we want $N = 2^n$ pairwise independent random variables, each of which is uniformly distributed in $\{0,1\}^m = [M]$. (Throughout this survey, we will often use the convention that a lowercase letter equals the logarithm (base 2) of the corresponding capital letter.) The naive approach is to repeat the above algorithm for the individual bits m times. This uses roughly $n \cdot m = (\log M)(\log N)$ initial random bits, which is no longer logarithmic in N if M is nonconstant. Below we will see that we can do much better. But first some definitions.

A sequence 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 3.21 (pairwise independent hash functions). A family (i.e., multiset) of functions $\mathcal{H} = \{h : [N] \to [M]\}$ is *pairwise independent* if the following two conditions hold when $H \stackrel{R}{\leftarrow} \mathcal{H}$ is a function chosen uniformly at random from $\mathcal{H}$:

(1) $\forall x \in [N]$, the random variable $H(x)$ is uniformly distributed in $[M]$.
(2) $\forall x_1 \neq x_2 \in [N]$, the random variables $H(x_1)$ and $H(x_2)$ are independent.

Equivalently, we can combine the two conditions and require that

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

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:

Definition 3.22. A family of functions $\mathcal{H} = \{h : [N] \to [M]\}$ is *explicit* if given the description of h and $x \in [N]$, the value $h(x)$ can be computed in time $\mathrm{poly}(\log N, \log M)$.

Pairwise independent hash functions are sometimes referred to as *strongly 2-universal hash functions*, to contrast with the weaker notion of *2-universal hash functions*, which requires only that $\Pr[H(x_1) = H(x_2)] \leq 1/M$ for all $x_1 \neq x_2$. (Note that this property is all we needed for the deterministic MaxCut algorithm, and it allows for a small savings in that we can also include $S = \emptyset$ in Construction 3.18.)

Below we present another construction of a pairwise independent family.

Construction 3.23 (pairwise independent hash functions from linear maps). Let $\mathbb{F}$ be a finite field. Define the family of functions $\mathcal{H} = \{ h_{a,b} : \mathbb{F} \to \mathbb{F} \}_{a,b \in \mathbb{F}}$ where $h_{a,b}(x) = ax + b$.

Proposition 3.24. The family of functions $\mathcal{H}$ in Construction 3.23 is pairwise independent.

Proof. 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 such line containing the points (x_1, y_1) and (x_2, y_2) (namely, the line with slope $a = (y_1 - y_2)/(x_1 - x_2)$ and y-intercept $b = y_1 - ax_1$). 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 $1/|\mathbb{F}|^2$. □

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 $h_{a,b} \xleftarrow{R} \mathcal{H}$. Compare this to $|\mathbb{F}| \log |\mathbb{F}|$ bits required to choose a truly random function, and $(\log |\mathbb{F}|)^2$ bits for repeating the construction of Proposition 3.19 for each output bit.

Note that evaluating the functions of Construction 3.23 requires a description of the field $\mathbb{F}$ that enables us to perform addition and multiplication of field elements. Thus we take a brief aside to review the complexity of constructing and computing in finite fields.

Remark 3.25 (constructing finite fields). Recall that for every prime power $q = p^k$ there is a field $\mathbb{F}_q$ (often denoted $\mathrm{GF}(q)$) of size q, and this field is unique up to isomorphism (renaming elements). The prime p is called the *characteristic* of the field. $\mathbb{F}_q$ has a description of length $O(\log q)$ enabling addition, multiplication, and division to be performed in polynomial time (i.e., time $\mathrm{poly}(\log q)$). (This description is simply an irreducible polynomial f of degree k over $\mathbb{F}_p = \mathbb{Z}_p$.) To construct this description, we first need to determine the characteristic p; finding a prime p of a desired bitlength ℓ can be done probabilistically in time $\mathrm{poly}(\ell) = \mathrm{poly}(\log p)$ and deterministically in time

$\text{poly}(2^\ell) = \text{poly}(p)$. Then given p and k, a description of $\mathbb{F}_q$ (for $q = p^k$) can be found probabilistically in time $\text{poly}(\log p, k) = \text{poly}(\log q)$ and deterministically in time $\text{poly}(p, k)$. Note that for both steps, the deterministic algorithms are exponential in the bitlength of the characteristic p. Thus, for computational purposes, a convenient choice is often to instead take $p = 2$ and k large, in which case everything can be done deterministically in time $\text{poly}(k) = \text{poly}(\log q)$.

Using a finite fields of size 2^n as suggested above, we obtain 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 $\{0,1\}^n$ 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 $m + n$ random bits by using $(ax)|_m + b$ where $b \in \{0,1\}^m$ instead of $(ax + b)|_m$. Summarizing:

Theorem 3.26. 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.

Problem 3.5 shows that $\max\{m,n\} + m$ random bits is essentially optimal.

3.5.3 Hash Tables

The original motivating application for pairwise independent functions was for hash tables. Suppose we want to store a set $S \subset [N]$ 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 $h : [N] \to [M]$ for some M to be determined later. Let T be a table of size M, initially empty. 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 be well-defined, 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 of size at most K, the probability that there are any collisions is

$$\begin{aligned}\Pr[\exists\ x \neq y \text{ s.t. } H(x) = H(y)] &\leq \sum_{x \neq y \in S} \Pr[H(x) \\ &= H(y)] \leq \binom{K}{2} \cdot \frac{1}{M} < \varepsilon\end{aligned}$$

for $M = K^2/\varepsilon$. Notice that the above analysis does not require H to be a completely random function; it suffices that H be pairwise independent (or even 2-universal). Thus using Theorem 3.26, we can generate and store H using $O(\log N)$ random bits. The storage required for the table T is $O(M \log N) = O(K^2 \log N)$ bits, which is much smaller than N when $K = N^{o(1)}$. Note that to uniquely represent a set of size K, we need space at least $\log \binom{N}{K} = \Omega(K \log N)$ (when $K \leq N^{0.99}$). In fact, there is a data structure achieving a matching space upper bound of $O(K \log N)$, which works by taking $M = O(K)$ in the above construction and using additional hash functions to separate the (few) collisions that will occur.

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 be preferable to show that some explicit family of hash function works for the application with similar performance. In many cases (such as the one above), it can be shown that pairwise independence (or k-wise independence, as discussed below) suffices.

3.5.4 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 this can be done using $O(k)$ independent repetitions (by a Chernoff Bound). If the algorithm originally used m random bits, then we use $O(km)$ random bits after error reduction. Here we will see how to reduce the number of random bits required for error reduction by doing repetitions that are only pairwise independent.

To analyze this, we will need an analogue of the Chernoff Bound (Theorem 2.21) that applies to averages of pairwise independent random variables. This will follow from Chebyshev's Inequality, which bounds the deviations of a random variable X from its mean μ in terms its *variance* $\mathrm{Var}[X] = \mathrm{E}[(X-\mu)^2] = \mathrm{E}[X^2] - \mu^2$.

Lemma 3.27 (Chebyshev's Inequality). If X is a random variable with expectation μ, then

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

Proof. Applying Markov's Inequality (Lemma 2.20) to the random variable $Y = (X-\mu)^2$, we have:

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

We now use this to show that a sum of pairwise independent random variables is concentrated around its expectation.

Proposition 3.28 (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

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{E}\left[(X-\mu)^2\right] \\
&= \frac{1}{t^2} \cdot \mathrm{E}\left[\left(\sum_i (X_i - \mu_i)\right)^2\right] \\
&= \frac{1}{t^2} \cdot \sum_{i,j} \mathrm{E}[(X_i - \mu_i)(X_j - \mu_j)] \\
&= \frac{1}{t^2} \cdot \sum_i \mathrm{E}[(X_i - \mu_i)^2] \quad \text{(by pairwise independence)} \\
&= \frac{1}{t^2} \cdot \sum_i \mathrm{Var}[X_i] \\
&\leq \frac{1}{t}
\end{aligned}$$

Now apply Chebyshev's Inequality. □

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

Error Reduction. Proposition 3.28 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 $h : \{0,1\}^{k+O(1)} \to \{0,1\}^m$ at random from a pairwise independent family, and running the algorithm using coin tosses $h(x)$ for all $x \in \{0,1\}^{k+O(1)}$. This requires $m + \max\{m, k + O(1)\} = O(m+k)$ random bits.

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

Note that we 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 constant error that uses m random bits, we have another **BPP** algorithm that uses $O(m + \log n) = O(m)$ random bits and has error $1/\text{poly}(n)$. That is, we can go from constant to inverse-polynomial error only paying a constant factor in randomness. (In fact, it turns out there is a way to achieve this with *no* penalty in randomness; see Problem 4.6.)

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

In Section 2.3.1, 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 (by Proposition 3.28). To generate t pairwise independent samples of m bits each, we need $O(m + \log t) = O(m + \log(1/\varepsilon) + \log(1/\delta))$ truly random bits.

	Number of Samples	Number of Random Bits
Truly Random Sample	$O((1/\varepsilon^2) \cdot \log(1/\delta))$	$O(m \cdot (1/\varepsilon^2) \cdot \log(1/\delta))$
Pairwise Independent Repetitions	$O((1/\varepsilon^2) \cdot (1/\delta))$	$O(m + \log(1/\varepsilon) + \log(1/\delta))$

Both of these sampling algorithms have a natural restricted structure. First, they choose all of their queries to the oracle f nonadaptively, based solely on their coin tosses and not based on the answers to previous queries. Second, their output is simply the average of the queried

values, whereas the original sampling problem does not constrain the output function. It is useful to abstract these properties as follows.

Definition 3.29. A *sampler* Samp for domain size M is given "coin tosses" $x \stackrel{R}{\leftarrow} [N]$ and outputs a sequence of samples $z_1, \ldots, z_t \in [M]$. We say that $\mathrm{Samp} : [N] \to [M]^t$ is a (δ, ε) *averaging sampler* if for every function $f : [M] \to [0,1]$, we have

$$\Pr_{(z_1,\ldots,z_t) \stackrel{R}{\leftarrow} \mathrm{Samp}(U_{[N]})} \left[\frac{1}{t} \sum_{i=1}^{t} f(z_i) > \mu(f) + \varepsilon \right] \leq \delta. \tag{3.3}$$

If Inequality 3.3 only holds for f with (boolean) range $\{0,1\}$, we call Samp a *boolean averaging sampler*. We say that Samp is *explicit* if given $x \in [N]$ and $i \in [t]$, $\mathrm{Samp}(x)_i$ can be computed in time $\mathrm{poly}(\log N, \log t)$.

We note that, in contrast to the Chernoff Bound (Theorem 2.21) and the Pairwise Independent Tail Inequality (Proposition 3.28), this definition seems to only provide an error guarantee in one direction, namely that the sample average does not significantly *exceed* the global average (except with small probability). However, a guarantee in the other direction also follows by considering the function $1 - f$. Thus, up to a factor of 2 in the failure probability δ, the above definition is equivalent to requiring that $\Pr[|(1/t) \cdot \sum_i f(z_i) - \mu(f)| > \varepsilon] \leq \delta$. We choose to use a one-sided guarantee because it will make the connection to list-decodable codes (in Section 5) slightly cleaner.

Our pairwise-independent sampling algorithm can now be described as follows:

Theorem 3.30 (Pairwise Independent Sampler). For every $m \in \mathbb{N}$ and $\delta, \varepsilon \in [0,1]$, there is an explicit (δ, ε) averaging sampler $\mathrm{Samp} : \{0,1\}^n \to (\{0,1\}^m)^t$ using $n = O(m + \log(1/\varepsilon) + \log(1/\delta))$ random bits and $t = O(1/(\varepsilon^2 \delta))$ samples.

As we will see in subsequent sections, averaging samplers are intimately related to the other pseudorandom objects we are studying (especially randomness extractors). In addition, some applications of samplers require samplers of this restricted form.

3.5.5 *k*-wise Independence

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

Definition 3.31 (k-wise independent hash functions). For $N, M, k \in \mathbb{N}$ such that $k \leq N$, 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 $H \xleftarrow{R} \mathcal{H}$.

Construction 3.32 (k-wise independence from polynomials). Let $\mathbb{F}$ be a finite field. Define the family of functions $\mathcal{H} = \{h_{a_0,a_1,\ldots,a_{k-1}} : \mathbb{F} \to \mathbb{F}\}$ where each $h_{a_0,a_1,\ldots,a_{k-1}}(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1}$ for $a_0, \ldots, a_{k-1} \in \mathbb{F}$.

Proposition 3.33. The family $\mathcal{H}$ given in Construction 3.32 is k-wise independent.

Proof. Similarly to the proof of Proposition 3.24, 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 g of degree at most $k-1$ such that $h(x_i) = g(x_i)$ for $i = 1, \ldots, k$. Then $h - g$ has at least k roots, and thus must be the zero polynomial. □

Corollary 3.34. 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 $\text{poly}(n, m, k)$.

---

k-wise independent hash functions have applications similar to those that pairwise independent hash functions have. The increased independence is crucial in derandomizing some algorithms. k-wise independent random variables also satisfy a tail bound similar to Proposition 3.28, with the key improvement being that the error probability vanishes linearly in $t^{k/2}$ rather than t; see Problem 3.8.

3.6 Exercises

Problem 3.1 (Derandomizing RP versus BPP*). Show that $\mathbf{prRP} = \mathbf{prP}$ implies that $\mathbf{prBPP} = \mathbf{prP}$, and thus also that $\mathbf{BPP} = \mathbf{P}$. (Hint: Look at the proof that $\mathbf{NP} = \mathbf{P} \Rightarrow \mathbf{BPP} = \mathbf{P}$.)

Problem 3.2 (Designs). Designs (also known as packings) are collections of sets that are nearly disjoint. In Section 7, we will see how they are useful in the construction of pseudorandom generators. Formally, a collection of sets $S_1, S_2, \ldots, S_m \subset [d]$ is called an (ℓ, a)-*design* (for integers $a \leq \ell \leq d$) if

- For all i, $|S_i| = \ell$.
- For all $i \neq j$, $|S_i \cap S_j| < a$.

For given ℓ, we'd like m to be large, a to be small, and d to be small. That is, we'd like to pack many sets into a small universe with small intersections.

(1) Prove that if $m \leq \binom{d}{a} / \binom{\ell}{a}^2$, then there exists an (ℓ, a)-design $S_1, \ldots, S_m \subset [d]$.
Hint: Use the Probabilistic Method. Specifically, show that if the sets are chosen randomly, then for every $S_1, \ldots, S_{i-1}$,

$$\mathop{\mathrm{E}}_{S_i}[\#\{j < i : |S_i \cap S_j| \geq a\}] < 1.$$

(2) Conclude that for every constant $\gamma > 0$ and every $\ell, m \in \mathbb{N}$, there exists an (ℓ, a)-design $S_1, \cdots, S_m \subset [d]$ with $d = O(\frac{\ell^2}{a})$ and $a = \gamma \cdot \log m$. In particular, setting $m = 2^\ell$, we fit exponentially many sets of size ℓ in a universe of size $d = O(\ell)$ while keeping the intersections an arbitrarily small fraction of the set size.
(3) Using the Method of Conditional Expectations, show how to construct designs as in Parts 1 and 2 *deterministically* in time $\text{poly}(m, d)$.

Problem 3.3 (More Pairwise Independent Families).

(1) (matrix-vector family) For an $n \times m$ $\{0,1\}$-matrix A and $b \in \{0,1\}^n$, define a function $h_{A,b} : \{0,1\}^m \to \{0,1\}^n$ by $h_{A,b}(x) = (Ax + b) \bmod 2$. (The "mod 2" is applied componentwise.) Show that $\mathcal{H}_{m,n} = \{h_{A,b}\}$ is a pairwise independent family. Compare the number of random bits needed to generate a random function in $\mathcal{H}_{m,n}$ to Construction 3.23.
(2) (Toeplitz matrices) A is a *Toeplitz matrix* if it is constant on diagonals, i.e., $A_{i+1,j+1} = A_{i,j}$ for all i, j. Show that even if we restrict the family $\mathcal{H}_{m,n}$ in Part 1 to only include $h_{A,b}$ for Toeplitz matrices A, we still get a pairwise independent family. How many random bits are needed now?

Problem 3.4 (Almost Pairwise Independence). A family of functions $\mathcal{H}$ mapping domain $[N]$ to range $[M]$ is *ε-almost pairwise independent*[3] if for every $x_1 \neq x_2 \in [N]$, $y_1, y_2 \in [M]$, we have

$$\Pr_{H \xleftarrow{R} \mathcal{H}} [H(x_1) = y_1 \text{ and } H(x_2) = y_2] \leq \frac{1 + \varepsilon}{M^2}.$$

[3] Another common definition of ε-almost pairwise independence requires instead that for every $x_1 \neq x_2 \in [N]$, if we choose a random hash function $H \xleftarrow{R} \mathcal{H}$, the random variable $(H(x_1), H(x_2))$ is ε-close to two uniform and independent elements of $[M]$ in statistical difference (as defined in Section 6). The two definitions are equivalent up to a factor of M^2 in the error parameter ε.

(1) Show that there exists a family $\mathcal{H}$ of ε-almost pairwise independent functions from $\{0,1\}^n$ to $\{0,1\}^m$ such that choosing a random function from $\mathcal{H}$ requires only $O(m + \log n + \log(1/\varepsilon))$ random bits (as opposed to $O(m+n)$ for exact pairwise independence). (Hint: First consider domain $\mathbb{F}^{d+1}$ for an appropriately chosen finite field $\mathbb{F}$ and $d \in \mathbb{N}$, and look at maps of the form $h = g \circ f_a$, where g comes from some pairwise independent family and $f_a : \mathbb{F}^{d+1} \to \mathbb{F}$ is defined by $f_a(x_0, \ldots, x_d) = x_0 + x_1 a + x_2 a^2 + \cdots + x_d a^d$.)

(2) Give a deterministic algorithm that on input an N-vertex, M-edge graph G (with no self-loops), finds a cut of size at least $(1/2 - o(1)) \cdot M$ in time $M \cdot \text{polylog}(N)$ and space $O(\log M)$ (thereby improving the $M \cdot \text{poly}(N)$ running time of Algorithm 3.20).

Problem 3.5 (Size Lower Bound for Pairwise Independent Families). Let $\mathcal{H} = \{h : [N] \to [M]\}$ be a pairwise independent family of functions.

(1) Prove that if $N \geq 2$, then $|\mathcal{H}| \geq M^2$.

(2) Prove that if $M = 2$, then $|\mathcal{H}| \geq N + 1$. (Hint: based on $\mathcal{H}$, construct a sequence of orthogonal vectors $v_x \in \{\pm 1\}^{|\mathcal{H}|}$ parameterized by $x \in [N]$.)

(3) More generally, prove that for arbitrary M, we have $|\mathcal{H}| \geq N \cdot (M-1) + 1$. (Hint: for each $x \in [N]$, construct $M-1$ linearly independent vectors $v_{x,y} \in \mathbb{R}^{|\mathcal{H}|}$ such that $v_{x,y} \perp v_{x',y'}$ if $x \neq x'$.)

(4) Deduce that for $N = 2^n$ and $M = 2^m$, selecting a random function from $\mathcal{H}$ requires at least $\max\{n, m\} + m$ random bits.

Problem 3.6 (Frequency Moments of Data Streams). Given one pass through a huge "stream" of data items $(a_1, a_2, \ldots, a_k)$, where each

$a_i \in \{0,1\}^n$, we want to compute statistics on the distribution of items occurring in the stream while using small space (not enough to store all the items or maintain a histogram). In this problem, you will see how to compute the *second frequency moment* $f_2 = \sum_a m_a^2$, where $m_a = \#\{i : a_i = a\}$.

The algorithm works as follows: Before receiving any items, it chooses t random *4-wise independent* hash functions $H_1, \ldots, H_t : \{0,1\}^n \to \{+1,-1\}$, and sets counters $X_1 = X_2 = \cdots = X_t = 0$. Upon receiving the ith item a_i, it adds $H_j(a_i)$ to counter X_j. At the end of the stream, it outputs $Y = (X_1^2 + \cdots + X_t^2)/t$.

Notice that the algorithm only needs space $O(t \cdot n)$ to store the hash functions H_j and space $O(t \cdot \log k)$ to maintain the counters X_j (compared to space $k \cdot n$ to store the entire stream, and space $2^n \cdot \log k$ to maintain a histogram).

(1) Show that for every data stream $(a_1, \ldots, a_k)$ and each j, we have $\mathrm{E}[X_j^2] = f_2$, where the expectation is over the choice of the hash function H_j.
(2) Show that $\mathrm{Var}[X_j^2] \leq 2f_2^2$.
(3) Conclude that for a sufficiently large constant t (independent of n and k), the output Y is within 1% of f_2 with probability at least 0.99.
(4) Show how to decrease the error probability to δ while only increasing the space by a factor of $\log(1/\delta)$.

Problem 3.7 (Improved Pairwise Independent Tail Inequality).

(1) Show that if X is a random variable taking values in $[0,1]$ and $\mu = \mathrm{E}[X]$, we have $\mathrm{Var}[X] \leq \mu \cdot (1-\mu)$.
(2) Improve the Pairwise Independent Tail Inequality (Proposition 3.28) to show that if X is the average of t pairwise independent random variables taking values in $[0,1]$ and $\mu = \mathrm{E}[X]$, then $\Pr[|X - \mu| \geq \varepsilon] \leq \mu \cdot (1-\mu)/(t \cdot \varepsilon^2)$. In particular, for $t = O(1/\mu)$, we have $\Pr[.99\mu \leq X \leq 1.01\mu] \geq 0.99$.

Problem 3.8 (k-wise Independent Tail Inequality). Let X be the average of t k-wise independent random variables for an even integer k, and let $\mu = \mathrm{E}[X]$. Prove that

$$\Pr[|X - \mu| \geq \varepsilon] \leq \left(\frac{k^2}{4t\varepsilon^2}\right)^{k/2}.$$

(Hint: show that all terms in the expansion of $\mathrm{E}[(X-\mu)^k] = \mathrm{E}[(\sum_i (X_i - \mu_i))^k]$ that involve more than $k/2$ variables X_i are zero.) Note that for fixed k, this probability decays like $O(1/(t\varepsilon^2))^{k/2}$, improving the $1/(t\varepsilon^2)$ bound in pairwise independence when $k > 2$.

Problem 3.9 (Hitting Samplers). A function $\mathrm{Samp} : [N] \to [M]^t$ is a (δ, ε) *hitting sampler* if for every set $T \subset [M]$ of density greater than ε, we have

$$\Pr_{(z_1,\ldots,z_t) \stackrel{R}{\leftarrow} \mathrm{Samp}(U_{[N]})} [\exists i \quad z_i \in T] \geq 1 - \delta.$$

(1) Show that every (δ, ε) averaging sampler is a (δ, ε) hitting sampler.
(2) Show that if we only want a hitting sampler, the number of samples in Theorem 3.30 can be reduced to $O(1/(\varepsilon\delta))$. (Hint: use Problem 3.7.)
(3) For which subset of **BPP** algorithms are hitting samplers useful for doing randomness-efficient error reduction?

3.7 Chapter Notes and References

The Time Hierarchy Theorem was proven by Hartmanis and Stearns [200]; proofs can be found in any standard text on complexity theory, for example, [32, 161, 366]. Adleman [3] showed that every language in **RP** has polynomial-sized circuits (cf., Corollary 3.12), and this was generalized to **BPP** by Gill. Pippenger [310] showed the equivalence between having polynomial-sized circuits and **P/poly**

(Fact 3.11). The general definition of complexity classes with advice (Definition 3.10) is due to Karp and Lipton [235], who explored the relationship between nonuniform lower bounds and uniform lower bounds. A $5n - O(n)$ circuit-size lower bound for an explicit function (in **P**) was given by Iwama et al. [218, 254].

The existence of universal traversal sequences (Example 3.8) was proven by Aleliunas et al. [13], who suggested finding an explicit construction (Open Problem 3.9) as an approach to derandomizing the logspace algorithm for Undirected S-T Connectivity. For the state of the art on these problems, see Section 4.4. An conjectured deterministic **NC** algorithm for Perfect Matching (derandomizing Algorithm 2.7 in a different way than Open Problem 3.7) is given in [4].

Theorem 3.14 is due to Sipser [364], who proved that **BPP** is in the fourth level of the polynomial-time hierarchy; this was improved to the second level by Gács. Our proof of Theorem 3.14 is due to Lautemann [256]. Problem 3.1 is due to Buhrman and Fortnow [85]. For more on nondeterministic computation and nonuniform complexity, see textbooks on computational complexity, such as [32, 161, 366].

The Method of Conditional Probabilities was formalized and popularized as an algorithmic tool in the work of Spencer [370] and Raghavan [316]. Its use in Algorithm 3.17 for approximating Max-Cut is implicit in [279]. For more on this method, see the textbooks [25, 291].

A more detailed treatment of pairwise independence (along with a variety of other topics in pseudorandomness and derandomization) can be found in the survey by Luby and Wigderson [281]. The use of limited independence in computer science originates with the seminal papers of Carter and Wegman [93, 417], which introduced the notions of universal, strongly universal (i.e., k-wise independent), and almost strongly universal (i.e., almost k-wise independent) families of hash functions. The pairwise independent and k-wise independent sample spaces of Constructions 3.18, 3.23, and 3.32 date back to the work of Lancaster [255] and Joffe [222, 223] in the probability literature, and were rediscovered several times in the computer science literature. The construction of pairwise independent hash functions from Part 1 of Problem 3.3 is due to Carter and Wegman [93] and Part 2 is implicit

in [408, 168]. The size lower bound for pairwise independent families in Problem 3.5 is due to Stinson [376], based on the Plackett–Burman bound for orthogonal arrays [312]. The construction of almost pairwise independent families in Problem 3.4 is due to Bierbrauer et al. [67] (though the resulting parameters follow from the earlier work of Naor and Naor [296]).

The application to hash tables from Section 3.5.3 is due to Carter and Wegman [93], and the method mentioned for improving the space complexity to $O(K \log N)$ is due to Fredman, Komlos, and Szemerédi [142]. The problem of randomness-efficient error reduction (sometimes called "deterministic amplification") was first studied by Karp, Pippenger, and Sipser [234], and the method using pairwise independence given in Section 3.5.4 was proposed by Chor and Goldreich [97]. The use of pairwise independence for derandomizing algorithms was pioneered by Luby [278]; Algorithm 3.20 for MaxCut is implicit in [279]. The notion of averaging samplers was introduced by Bellare and Rompel [57] (under the term "oblivious samplers"). For more on samplers and averaging samplers, see the survey by Goldreich [155]. Tail bounds for k-wise independent random variables, such as the one in Problem 3.8, can be found in the papers [57, 97, 350].

Problem 3.2 on designs is from [134], with the derandomization of Part 3 being from [281, 302]. Problem 3.6 on the frequency moments of data streams is due to Alon, Matias, and Szegedy [22]. For more on data stream algorithms, we refer to the survey by Muthukrishnan [295].