# 3

# Basic Derandomization Techniques

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

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

In the rest of this survey, we will turn towards *derandomization* — trying to remove the randomness from these algorithms. We will achieve this for some of the specific algorithms we studied, and also attack the larger question of whether *all* efficient randomized algorithms can be derandomized, e.g. does **BPP** = **P**? **RL** = **L**? **RNC** = **NC**?

In this chapter, 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 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 Chapters 4+).

## 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]

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

---

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 vs. nondeterministic time.)

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

---

**Proposition 3.2.** $\mathbf{BPP} \subseteq \mathbf{EXP}$.

---

*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 an upper bound, $A$ uses at most $t(n)$ random bits. *Thus we can view $A$ as a deterministic algorithm on two inputs — its regular input $x \in \{0,1\}^n$ and its coin tosses $r \in \{0,1\}^{t(n)}$.* (This view of a randomized algorithm is useful throughout the study of pseudorandomness.) We'll write $A(x; r)$ for $A$'s output. Then:

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

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

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 $r(n)$ random bits, then $L \in \mathbf{DTIME}(t(n) \cdot 2^{r(n)})$. In particular, if $t(n) = \text{poly}(n)$ and $r(n) = O(\log n)$, 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 Chapter **??**. However, to date, Proposition 3.2 remains the best unconditional upper-bound we have on the deterministic time-complexity of **BPP**.

[1] Often $\mathbf{DTIME}(\cdot)$ is written as $\mathbf{TIME}(\cdot)$, but we include the **D** to emphasize the it refers to deterministic rather than randomized algorithms.

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

## 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 $\alpha \stackrel{\text{R}}{\leftarrow} S_m^{m^2}$ 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 $\det(A^G(\alpha)) \neq 0$ 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 the previous lecture, we know that a random walk of $\mathrm{poly}(n, d)$ 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* $w \in \{1, 2, \ldots, d\}^{\mathrm{poly}(n,d)}$ 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 towards 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 $\mathbf{C}$ be a class of languages, and $\mathbf{a} : \mathbb{N} \to \mathbb{N}$ be a function. Then $\mathbf{C}/\mathbf{a}$ 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 $|\alpha_n| \le a(n)$, such that $x \in L \Leftrightarrow (x, \alpha_{|x|}) \in L'$. The $\alpha$'s are called the *advice* strings.

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

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

**Fact 3.11.** $L \in \mathbf{P}/\mathbf{poly}$ 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) $|C_n| \le p(n)$.

We refer to $\mathbf{P}/\mathbf{poly}$ 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 $\alpha_n$), 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 $\mathbf{P}/\mathbf{poly}$ contains some undecidable problems,[2] people generally believe that $\mathbf{NP} \not\subseteq \mathbf{P}/\mathbf{poly}$, and 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 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:

[2] Consider the unary version of halting problem, the advice string $\alpha_n$ is simply a bit that tells us whether the $n$'th Turing machine halts or not.

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

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

### 3.3 Nondeterminism

Although physically unrealistic, nondeterminism has proved be a very useful resource in the study of computational complexity (e.g. leading to the class **NP**). Thus it is natural 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.** $\mathbf{RP} \subseteq \mathbf{NP}$.

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

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 $\mathbf{BPP} = \mathbf{NEXP}$! 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 $P$,

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

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 $Z_x \subset \{0,1\}^m$ be the set of coin tosses $r$ for which $A(x;r) = 0$. We will show that if $x$ is in $L$, there exist $m$ points in $\{0,1\}^m$ such that no "shift" (or "translation") of $Z_x$ covers all the points. (Notice that this is a $\exists\forall$ statement.) Intuitively, this should be possible because $Z_x$ is an exponentially small fraction of $\{0,1\}^m$. On the other hand if $x \notin L$, then for any $m$ points in $\{0,1\}^m$, we will show that there is a "shift" of $Z_x$ that covers all the points. Intuitively, this should be possible because $Z_x$ covers all but an exponentially small fraction of $\{0,1\}^m$.

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

$$\begin{aligned} x \in L \quad &\Rightarrow \quad \exists r_1, r_2, \ldots, r_m \in \{0,1\}^m \forall s \in \{0,1\}^m \quad \neg \bigwedge_{i=1}^{m} (r_i \in Z_x \oplus s) \\ &\Leftrightarrow \quad \exists r_1, r_2, \ldots, r_m \in \{0,1\}^m \, \forall s \in \{0,1\}^m \quad \neg \bigwedge_{i=1}^{m} (A(x; r_i \oplus s) = 0)\,; \end{aligned}$$

$$\begin{aligned} x \notin L &\Rightarrow \forall r_1, r_2, \ldots, r_m \in \{0,1\}^m \, \exists s \in \{0,1\}^m \quad \bigwedge_{i=1}^{m} (r_i \in Z_x \oplus s) \\ &\Leftrightarrow \forall r_1, r_2, \ldots, r_m \in \{0,1\}^m \, \exists s \in \{0,1\}^m \quad \bigwedge_{i=1}^{m} \left(A(x; r_i \oplus s) = 0\right). \end{aligned}$$

We prove both parts by the Probabilistic Method.

$x \in L$: Choose $R_1, R_2, \ldots, R_m \stackrel{\text{R}}{\leftarrow} \{0,1\}^m$. Then, for every fixed $s$, $Z_x$ and hence $Z_x \oplus s$ contains less than a $2^{-n}$ fraction of points in $\{0,1\}^m$, so:

$$\begin{aligned} \forall i \quad & \Pr[R_i \in Z_x \oplus s] < 2^{-n} \\ \Rightarrow \quad & \Pr\left[\bigwedge_i (R_i \in Z_x \oplus s)\right] < 2^{-nm}. \\ \Rightarrow \quad & \Pr\left[\exists s \bigwedge_i (R_i \in Z_x \oplus s)\right] < 2^m \cdot 2^{-nm} < 1. \end{aligned}$$

Thus there exist $r_1, \ldots, r_m$ such that $\forall s \, \neg \bigwedge_i (r_i \in Z_x \oplus s)$, as desired.

$x \notin L$: Let $r_1, r_2, \ldots, r_m$ be arbitrary, and choose $S \stackrel{\text{R}}{\leftarrow} \{0,1\}^m$ at random. Now $Z_x$ and hence $Z_x \oplus r_i$ contains more than a $1 - 2^{-n}$ fraction of points, so:

$$\begin{aligned} \forall i \quad & \Pr[r_i \notin Z_x \oplus S] = \Pr[S \notin Z_x \oplus r_i] < 2^{-n} \\ \Rightarrow \quad & \Pr\left[\bigvee_i (r_i \notin Z_x \oplus S)\right] < m \cdot 2^{-n} < 1. \end{aligned}$$

Thus, for every $r_1, \ldots, r_m$, there exists $s$ such that $\bigwedge_i (r_i \in Z_x \oplus s)$, as desired.

□

Readers familiar with complexity theory will recognize the above proof as showing that **BPP** is contained in the 2nd level of the *polynomial-time hierarchy* (**PH**). In general, the $k$'th 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," i.e., 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 $1 \le i \le m$ 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, if $R_1, \ldots, R_m$ are uniform and independent random bits, then

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

(See Figure 3.4.1.) By averaging, there exists an $r_{i+1} \in \{0,1\}$ such that $P(r_1, r_2, \ldots, r_i, r_{i+1}) \ge$

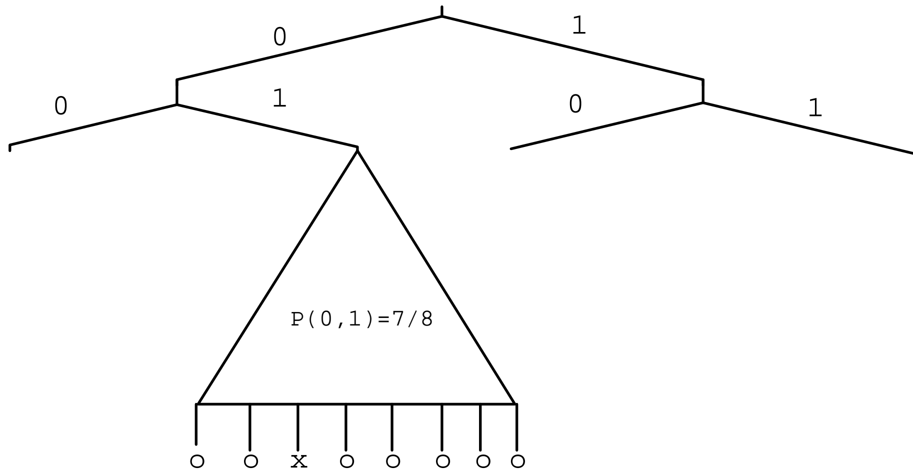


Fig. 3.1 An example of $P(r_1, r_2)$, where "o" at the leaf denotes a good path.

$P(r_1, r_2, \ldots, r_i)$. 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

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

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, often for *search* problems rather than decision problems, and where we measure not a boolean outcome (eg 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 MaxCut problem:

**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 $\mathrm{cut}(S,T) = \{\{u,v\} \in E : u \in S, v \in T\}$.

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

**Algorithm 3.16 (randomized MaxCut, rephrased).**
Input: a graph $G = ([n], E)$

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

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

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

to be the expected cut size when the 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, $e[\Lambda] \geq |E|/2$ (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 the quantity is not hard to compute. Let $S_i \stackrel{\text{def}}{=} \{j : j \leq i, r_j = 1\}$ (resp. $T_i \stackrel{\text{def}}{=} \{j : j \leq i, r_j = 0\}$) 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

$$e(r_1, r_2, \ldots, r_i) = |\mathrm{cut}(S_i, T_i)| + 1/2 \left( |\mathrm{cut}(S_i, U_i)| + |\mathrm{cut}(T_i, U_i)| + |\mathrm{cut}(U_i, U_i)| \right). \tag{3.2}$$

Note that $\mathrm{cut}(U_i, U_i)$ 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 (3.2) in its entirety. Observe, in (3.2), that whether $R_{i+1}$ equals 0 or 1 does not affect the *change* of the quantity in the parenthesis. That is,

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

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 $\mathrm{cut}(U, U)$ term but added to either the $\mathrm{cut}(S, U)$ or $\mathrm{cut}(T, U)$ term depending on whether $i+1$ is put in $S$ or $T$, respectively. (See Figure 3.4.2.) The edges with one endpoint being $i+1$ and the other being in $S_i \cup T_i$ will be removed from the $|\mathrm{cut}(S, U)| + |\mathrm{cut}(T, U)|$ 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 $|\mathrm{cut}(S, T)|$ term. This term increases by either $|\mathrm{cut}(\{i+1\}, T_i)|$ or $|\mathrm{cut}(\{i+1\}, S_i)|$ depending on whether we place vertex $i+1$ in $S$ or $T$, respectively. To summarize, we have

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

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

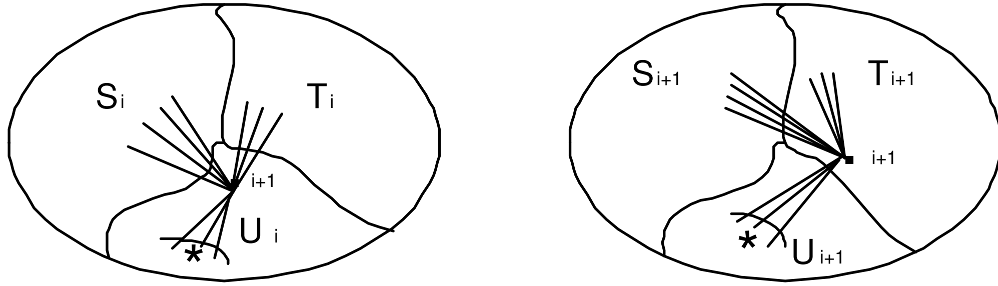


Fig. 3.2 Marked edges are subtracted in $\mathrm{cut}(U, U)$ but added back in $\mathrm{cut}(T, U)$. The total change is the unmarked edges.

---

**Algorithm 3.17 (deterministic MaxCut approximation).**
Input: A graph $G = ([n], E)$

On input $G = ([n], E)$,

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

## 3.5 Pairwise Independence

### 3.5.1 An Example

As our first motivating example, 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 $S \subseteq [k]$, let $R_S$ be the random variable $\oplus_{i \in S} B_i$.

**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 $S \neq T \subseteq [k]$. 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. 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).** 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 $b_i$'s, the derandomized algorithm still runs in $\mathrm{poly}(n)$ 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.

### 3.5.2 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. Below we will see that we can do much better. But first some definitions.

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 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 $H \stackrel{\mathrm{R}}{\leftarrow} \mathcal{H}$ 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

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

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 $\text{poly}(\log N, \log M)$.

---

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 $h_{a,b} \stackrel{\text{R}}{\leftarrow} \mathcal{H}$. 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. Recall that there is a (unique) finite field $\text{GF}(p^t)$ of size $p^t$ for every prime $p$ and $t \in \mathbb{N}$. It is known how to deterministically construct a description of such a field (i.e. an irreducible polynomial of degree $t$ over $\text{GF}(p) = \mathbb{Z}_p$) in time $\text{poly}(p, t)$. This satisfies our definition of explicitness when the prime $p$ (the *characteristic* of the field) is small, in particular when $p = 2$. 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 $m+n$ random bits (which turns out to be optimal) by using $(ax)|_m + b$ where $b \in \{0,1\}^m$ instead of $(ax+b)|_m$. Summarizing:

---

**Theorem 3.25.** 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.

---

### 3.5.3 Hash Tables

The original motivating application for pairwise independent functions was for hash tables. Suppose we want to store a set $S \subseteq [N]$ 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 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 $S$, the probability that there are any collisions is

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

for $M = |S|^2/\varepsilon$. 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 3.25, 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 (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 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 averages of pairwise independent random variables. This will follow from Chebychev's Inequality, which bounds the deviations of a random variable $X$ from its mean $\mu$ in terms its *variance* $\mathrm{Var}[X] = \mathrm{E}[(X-\mu)^2] = \mathrm{E}[X^2] - \mu^2$.

---

**Lemma 3.26 (Chebyshev's Inequality).** If $X$ is a random variable with expectation $\mu$, 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:

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

□

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

---

**Proposition 3.27 (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

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

Now apply Chebychev'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 3.27 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 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 $1/\text{poly}(n)$. 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 4.7.)

**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 $\varepsilon$.

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 $\delta$. 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 | $O(1/(\varepsilon^2\delta))$ | $O(m + \log(1/\varepsilon) + \log(1/\delta))$ |

### 3.5.5 $k$-wise Independence

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

---

**Definition 3.28 ($k$-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 $H \stackrel{\text{R}}{\leftarrow} \mathcal{H}$.

---

**Construction 3.29 ($k$-wise independence from polynomials).** Let $\mathbb{F}$ be a finite field. Define the family of functions $\mathcal{H} = \{h_{a_0,a_1,\ldots,a_k} : \mathbb{F} \to \mathbb{F}\}$ 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 $a, b \in \mathbb{F}$.

**Proposition 3.30.** The family $\mathcal{H}$ given in Construction 3.29 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. $\square$

**Corollary 3.31.** 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 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 3.27, with the key improvement being that the error probability vanishes linearly in $t^{k/2}$ rather than $t$.

## 3.6 Exercises

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

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

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

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

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

(2) Conclude that for every $\epsilon > 0$, there is a constant $c_\epsilon$ such that for all $\ell$, there is a design with $a \leq \epsilon\ell$, $m \geq 2^{\epsilon\ell}$, and $d \leq c_\epsilon \ell$. That is, in a universe of size $O(\ell)$, we can fit *exponentially many* sets of size $\ell$ whose intersections are an arbitrarily small constant fraction of $\ell$.
(3) Using the Method of Conditional Expectations, show how to construct designs as in Part 1 and *deterministically* in time $\mathrm{poly}(m, d)$.

---

---

**Problem 3.3.** (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 *2nd 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 $i$'th 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 .99.

---

---

**Problem 3.4.** (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?

---

## 3.7 Chapter Notes and References

The Time Hierarchy Theorem was proven by Hartmanis and Stearns [HS]; proofs can be found in any standard text on complexity theory, e.g. [Sip2, Gol6, AB]. Adleman [Adl] showed that every language in **RP** has polynomial-sized circuits (cf., Corollary 3.12), and Pippenger [Pip] 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 [KL], 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. [LR, IM].

The existence of universal traversal sequences (Example 3.8) was proven by Aleliunas et al. [AKL$^+$], 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.

Theorem 3.14 is due to Sipser [Sip1], who proved that **BPP** is the 4th level of the polynomial-time hierarchy; this was improved to the 2nd level by Gács. Our proof of Theorem 3.14 is due to Lautemann [Lau]. Problem 3.1 is due to Buhrman and Fortnow [BF]. For more on nondeterministic computation and nonuniform complexity, see textbooks on computational complexity, such as [Sip2, Gol6, AB].

The Method of Conditional Probabilities was formalized and popularized as an algorithmic tool in the work of Spencer [Spe] and Raghavan [Rag]. Its use in Algorithm 3.17 for approximating MAXCUT is implicit in Luby [Lub2]. For more on this method, see the textbooks [MR, AS].

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 [LW]. The use of pairwise independence in computer science originates with the seminal papers of Carter and Wegman [CW, WC], which introduced the notions of universal and strongly universal families of hash functions. The pairwise independent and $k$-wise independent sample spaces of Constructions 3.18, 3.23, and 3.29 date back to the work of Lancaster [Lan] and Joffe [Jof1, Jof2] in the probability literature, and were rediscovered several times in the computer science literature. . The constructions of pairwise independent hash functions from Problem 3.4 are due to Carter and Wegman [CW]. The application to hash tables from Section 3.5.3 is due to Carter and Wegman [CW], and the method mentioned for improving the space complexity to $O(|S| \log N)$ is due to Fredman, Komlós, and Szemerédi [FKS]. The problem of randomness-efficient error reduction (sometimes called "deterministic amplification") was first studied by Karp, Pippenger, and Sipser [KPS], and the method using pairwise independence given in Section 3.5.4 was proposed by Chor and Goldreich [CG]. The use of pairwise independence for derandomizing algorithms was pioneered by Luby [Lub1]; Algorithm 3.20 for MAXCUT is implicit in [Lub2]. Tail bounds for $k$-wise independent random variables can be found in the papers [CG, BR, SSS].

Problem 3.2 on designs is from [EFF], with the derandomization of Part 3 being from [NW, LW].

Problem 3.3 on the frequency moments of data streams is due to Alon, Mathias, and Szegedy [AMS]. For more on data stream algorithms, we refer to the survey by Muthukrishnan [Mut].