ENCODING ARGUMENTS*

Pat Morin, Wolfgang Mulzer, and Tommy Reddad

Abstract.

This expository article surveys "encoding arguments." In its most most basic form an encoding argument proves an upper bound on the probability of an event using the fact a uniformly random choice from a set of size n cannot be encoded with fewer than $\log_2 n$ bits on average.

We survey many applications of this basic argument, give a generalization of this argument for the case of non-uniform distributions, and give a rigorous justification for the use of non-integer codeword lengths in encoding arguments. These latter two results allow encoding arguments to be applied more widely and to produce tighter results.

Keywords: Encoding arguments, entropy, Kolmogorov complexity, incompressibility, analysis of algorithms, hash tables, random graphs, expanders, concentration inequalities, percolation theory.

*This research is partially funded by NSERC. W. Mulzer is supported by DFG Grants 3501/1 and 3501/2.

Contents

1 Introduction 1

2 Background 2

2.1 Basic Definitions, Prefix-free Codes and the Uniform Encoding Lemma . . . 2

2.2 Runs in Binary Strings . . . 4

2.3 A Note on Ceilings . . . 6

2.4 Encoding Sparse Bit Strings . . . 6

2.5 Binary Entropy . . . 7

2.6 Basic Chernoff Bound . . . 9

2.7 Factorials and Binomial Coefficients . . . 9

2.8 Encoding the Natural Numbers . . . 10

3 Applications of the Uniform Encoding Lemma 11

3.1 Graphs with no Large Clique or Independent Set . . . 11

3.2 Balls in Urns . . . 13

3.3 Linear Probing . . . 14

3.4 Cuckoo Hashing . . . 16

3.5 2-Choice Hashing . . . 18

3.6 Bipartite Expanders . . . 22

3.7 Permutations and Binary Search Trees . . . 23

3.7.1 Analysis of Insertion Sort . . . 24

3.7.2 Records . . . 25

3.7.3 The Height of a Random Binary Search Tree . . . 27

3.7.4 Hoare's Find Algorithm . . . 29

3.8 k-SAT and the Lovász Local Lemma . . . 31

4 The Non-Uniform Encoding Lemma and Shannon-Fano Codes **34**

5 Applications of the Non-Uniform Encoding Lemma **35**

5.1 Chernoff Bound 35

5.2 Percolation on the Torus 36

5.3 Triangles in $G_{n,p}$ 38

6 Encoding with Kraft's Condition **40**

7 Summary and Conclusions **42**

1 Introduction

There is no doubt that probability theory plays a fundamental role in computer science: Some of the fastest and simplest fundamental algorithms and data structures are randomized [24, 28]; average-case analysis of algorithms relies entirely on tools from probability theory [35]; and many difficult combinatorial questions have strikingly simple solutions using probabilistic arguments [1].

Unfortunately, many of these beautiful results present a challenge to most computer scientists because of the advanced mathematical concepts they rely on. For instance, the 2013 edition of ACM/IEEE Curriculum Guidelines for Undergraduate Degree Programs in Computer Science does not require a full course in probability theory [29, Page 50]. Indeed, the report recommends a total of 6 Tier-1 hours and 2 Tier-2 hours spent on discrete probability, as part of the discrete structures curriculum [29, Page 77].

In this expository paper, we survey applications of "encoding arguments" that transform the problem of upper-bounding the probability of a specific event, $\mathcal{E}$, into the problem of devising a code for the set of elementary events in $\mathcal{E}$. Such a problem could also be approached using a traditional probabilistic analysis or, since we are only concerned with finite spaces, by directly counting the size of $\mathcal{E}$. Of course, the probabilistic method offers many theoretical and intuitive advantages over direct counting, even though a probabilistic argument is only effectively a sophisticated rephrasing of a counting argument. Accordingly, an encoding argument offers its own set of advantages over an alternative proof technique, even though it also effectively is a rephrased form of counting. More specifically:

1. Encoding arguments are almost "probability-free." Except for applying a simple *Uniform Encoding Lemma*, there is no probability involved. In particular, there is no chance of common mistakes such as multiplying probabilities of non-independent events or (equivalently) multiplying expectations.

   The proof of the Uniform Encoding Lemma itself is trivial and the only probability it uses is the fact that, if a finite set X contains r special elements and we pick an element uniformly at random from X, then the probability of picking a special element is $r/|X|$.

2. Encoding arguments usually yield strong results; $\Pr\{\mathcal{E}\}$ typically decreases at least exponentially in the parameter of interest. Traditionally, these strong concentration results require (at least) careful calculations on probabilities of independent events and/or the application of concentration inequalities. The subject of concentration

inequalities is advanced enough to be the topic of entire textbooks [6, 11].

3. Encoding arguments are natural for computer scientists. They turn a probabilistic analysis problem into the problem of designing an efficient code—an algorithmic problem. Consider the following two problems:

   (a) Prove an upper-bound of $1/n^{\log n}$ on the probability that a random graph on n vertices contains a clique of size $k = \lceil 4\log n\rceil$.[1]

   (b) Design an encoding for graphs on n vertices so that a graph, G, that contains a clique of size $k = \lceil 4\log n\rceil$, is encoded using at most $\binom{n}{2} - \log^2 n$ bits. (Note: Your encoding and decoding algorithms don't have to be efficient, just correct.)

   Many computer science undergraduates would not know where to start on the first problem. Even a good student who realizes that they can use Boole's Inequality will still be stuck wrestling with the formula $\binom{n}{4\log n}2^{-\binom{4\log n}{2}}$.

Our motivation for this work is that encoding arguments are an easily accessible, yet versatile tool for answering many questions. Most of these arguments can be applied after learning almost no probability theory beyond the Encoding Lemma mentioned above.

The remainder of this article is organized as follows: In Section 2, we present necessary background, including the *Uniform Encoding Lemma*, which is the basis of most of our encoding arguments. In Section 3 we show how the Uniform Encoding Lemma can be applied to a variety of problems. In Section 4, we introduce a more general Non-Uniform Encoding Lemma that can handle a larger variety of applications, some of which are given in Section 5. Section 6 presents an alternative view of encoding arguments, justifying the use of non-integer codeword lengths. Section 7 summarizes and concludes with some directions for future research.

2 Background

This section presents the necessary background on prefix-free codes and binomial coefficients.

2.1 Basic Definitions, Prefix-free Codes and the Uniform Encoding Lemma

A finite length *binary string*, or *bit string*, is a finite (possibly empty) sequence of elements from $\{0,1\}$. For $k \in \mathbb{N}$, we denote by $\{0,1\}^k$ the set of all binary strings of length k, and by

[1] Since we are overwhelmingly concerned with binary encoding, we will agree now that the base of logarithms in $\log x$ is 2, except when explicitly stated otherwise.

x	$C(x)$
a	0
b	100
c	1010
d	1011
e	110
f	1111

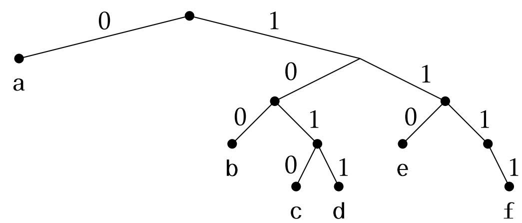


Figure 1: A prefix-free code for the set $X = \{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d},\mathrm{e},\mathrm{f}\}$ and the corresponding leaf-labelled binary tree (which can also be viewed as a partial prefix-free code for the set $\{\mathrm{a},\mathrm{b},\mathrm{c},\ldots,\mathrm{z}\}$).

$\{0,1\}^*$ the set of all finite strings. For a binary string, x, we use $|x|$ to denote the length of x. A binary string of length n will be called an n-bit string. Furthermore, we denote by $n_1(x)$ the number of 1-bits in x, and by $n_0(x)$ the number of zero bits. Given a (finite or countable) set X, a *code* $C\colon X \to \{0,1\}^*$ is a one-to-one function from X to the set of finite length binary strings. The elements of the range of C are called C's *codewords*. Often, there are some elements of the set X that are not of interest to us. In these cases, we consider partial codes. A *partial code* $C\colon X \nrightarrow \{0,1\}^*$ is a one-to-one partial function. When discussing partial codes we will use the convention that $|C(x)| = \infty$ if x is not in the domain of C.

A binary string x is called a *prefix* of another binary string y if there is some binary string z such that $xz = y$. A (partial) code C is *prefix-free* if, for every pair $x \neq y$ in the domain of C, the codeword $C(x)$ is not a prefix of the codeword $C(y)$. It can be helpful to visualize prefix-free codes as (rooted ordered) binary trees whose leaves are labelled with the elements of X. The codeword for a particular $x \in X$ is obtained by tracing the root-to-leaf path leading to x and outputting a 0 each time this path goes from a parent to its left child, and a 1 each time it goes to a right child. (See Figure 1.)

We claim that if C is prefix-free, then the number of C's codewords that have length at most k is not more than 2^k. To see this, observe that C can be modified into a code $\widehat{C}$, in which every codeword of length $\ell < k$ is extended—by appending $k-\ell$ zeros—so that it has length exactly k. The prefix-freeness of C ensures that $\widehat{C}$ is also prefix-free. The number of $\widehat{C}$'s codewords of length k is equal to the number of C's codewords of length at most k; since codewords are just binary strings, there are not more than 2^k of these.

Observe that every finite set X has a prefix-free code in which every codeword has length $\lceil \log|X| \rceil$. We simply enumerate the elements of X in some order $x_0, x_1, \ldots, x_{|X|-1}$ and assign to each x_i the binary representation of i (padded with leading zeros), which has length $\lceil \log|X| \rceil$, since $i \in \{0,\ldots,|X|-1\}$. We call this encoding the *fixed-length code* for X, and we will use it implicitly in many arguments.

$$\begin{array}{rcccccccccc} & & & & \multicolumn{4}{c}{t} & & & \\ x = & \star & \cdots & \star & 1 & 1 & & 1 & \star & \cdots & \star \\ & x_1 & \cdots & x_{i-1} & x_i & x_{i+1} & \cdots & x_{i+t-1} & x_{i+t} & \cdots & x_n \end{array}$$

$$C(x) = \underbrace{\qquad i \qquad}_{\lceil \log n \rceil}\ \underbrace{x_1 \quad \cdots \quad x_{i-1} \quad x_{i+t} \quad \cdots \quad x_n}_{n-t}$$

Figure 2: Illustration of Theorem 1 and its proof.

Given a (finite or countable) set X, a *probability distribution* $p : X \to [0,1]$ on X is a function with $\sum_{x \in X} p(x) = 1$. We sometimes write p_x instead of $p(x)$. The *uniform distribution* is given by $p_x = 1/|X|$ for all $x \in X$. Fix a number $n \in \mathbb{N}$ and a parameter $\alpha \in [0,1]$. A probability distribution on $X = \{0,1\}^n$ that is of particular interest to us is the *Bernoulli distribution* with parameter α, denoted Bernoulli(α). In this distribution, a random bit string $x \in \{0,1\}^n$ is sampled by setting each bit to 1 with probability α and to 0 with probability $1-\alpha$, independently of the other bits.

The following lemma provides the foundation on which this survey is built. The lemma is folklore, but as we will see in the following sections, it has an incredibly wide range of applications and can lead to surprisingly powerful results.

Lemma 1 (Uniform Encoding Lemma)**.** *Let X be a finite set, and let $C\colon X \nrightarrow \{0,1\}^*$ be a partial prefix-free code. If an element $x \in X$ is chosen uniformly at random, then*

$$\Pr\{|C(x)| \le \log|X| - s\} \le 2^{-s} \ .$$

Proof. Let $k = \lfloor \log|X| - s \rfloor$ and recall that C has at most 2^k codewords of length at most k. Since C is one-to-one each such codeword has at most one preimage in X. Since x is chosen uniformly at random from X, the probability that it is the preimage of one of these short codewords is at most

$$\frac{2^k}{|X|} \le \frac{2^{\log|X|-s}}{|X|} = 2^{-s} \ . \qquad \square$$

2.2 Runs in Binary Strings

As a warm-up exercise to illustrate the use of the Uniform Encoding Lemma we will show that a random n-bit string is unlikely to contain a run of significantly more than $\log n$ one bits. (See Figure 2.)

Theorem 1. *Let $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ be chosen uniformly at random and let $t = \lceil \lceil \log n \rceil + s \rceil$.*

Then, the probability that there exists an $i \in \{1,\dots,n-t+1\}$ *such that* $x_i = x_{i+1} = \cdots = x_{i+t-1} = 1$ *is at most* 2^{-s}.

Proof. We will prove this theorem by constructing a partial prefix-free code for strings having a run of t or more ones. For such a string $x = (x_1,\dots,x_n)$ let i be the minimum index such that $x_i = x_{i+1} = \cdots = x_{i+t-1} = 1$. The codeword $C(x)$ for x is the binary string that consists of the ($\lceil \log n \rceil$-bit binary encoding of the) index i followed by the $n-t$ bits $x_1,\dots,x_{i-1},x_{i+t},\dots,x_n$. (See Figure 2.)

Observe that $C(x)$ has length

$$\lceil \log n \rceil + n - t \le n - s \ .$$

For any such x, we can reconstruct $(x_1,\dots,x_n)$ from $C(x)$ by reading it from left to right. Indeed, the leftmost $\lceil \log n \rceil$ bits from $C(x)$ tell us, in binary, the value of an index i for which $x_i = x_{i+1} = \cdots = x_{i+t-1} = 1$; the following $n-t$ bits in sequence give us the values of the remaining unknown bits $x_1,x_2,\dots,x_{i-1},x_{i+t},x_{i+t+1},\dots,x_n$. Thus, C is a partial code whose domain is the set of binary strings of length n having a run of t or more ones. Also, C is prefix-free, since all codewords are unique and have the same length.

Now, x was chosen uniformly at random from a set of size 2^n. Therefore, by the Uniform Encoding Lemma, the probability that there exists any index $i \in \{1,\dots,n-t-1\}$ such that $x_i = x_{i+1} = \cdots = x_{i+t-1} = 1$ is at most

$$\Pr\{|C(x)| \le n - s\} \le 2^{-s} \ . \qquad \square$$

Simple as it is, the proof of Theorem 1 contains the main ideas used in most encoding arguments:

1. The arguments usually show that a particular *bad event* is unlikely. In Theorem 1 the bad event is the occurrence of a substring of t consecutive ones.
2. The code is a partial prefix-free code whose domain is the bad event, whose elements we call the *bad outcomes*. In this case, the code C is only capable of encoding strings containing a run of t consecutive ones, and a particular string containing such a run is a bad outcome.
3. The code usually begins with a concise description of the bad outcome, and is then followed by a straightforward encoding of the information that is not implied by the bad outcome. In Theorem 1, the bad outcome is completely described by the index i at which the run of t ones begins, and this implies that the bits $x_i,\dots,x_{i+t-1}$

are all equal to 1, so these bits do not need to be specified in the second part of the codeword.

2.3 A Note on Ceilings

Note that Theorem 1 also has an easy proof using the union bound: If we let $\mathcal{E}_i$ denote the event $x_i = x_{i+1} = \cdots = x_{i+t} = 1$, then

$$\begin{aligned}
\Pr\left\{\bigcup_{i=0}^{n-t-1} \mathcal{E}_i\right\} &\le \sum_{i=0}^{n-t-1} \Pr\{\mathcal{E}_i\} && \text{(using the union bound)} \\
&= \sum_{i=0}^{n-t-1} 2^{-t} && \text{(using the independence of the } x_i\text{'s)} \\
&\le n2^{-t} && \text{(the sum has } n-t \le n \text{ identical terms)} \\
&\le n2^{-\lceil \log n \rceil - s} && \text{(from the definition of } t\text{)} \\
&\le 2^{-s} \enspace .
\end{aligned}$$

This traditional proof also works with the sometimes smaller value $t = \lceil \log n + s \rceil$ (note the lack of a ceiling over the logarithmic term), in which case the final inequality becomes an equality.

In the encoding proof of Theorem 1, the ceiling in the expression for t is an artifact of encoding the integer i which is taken from a set of size n. When sketching an encoding argument, we think of this as requiring $\log n$ bits. Nonetheless, when the time comes to carefully write down a proof we include a ceiling over this term since bits are a discrete quantity.

In Section 6, however, we will formally justify that the informal intuition we use in blackboard proofs is actually valid; we can think of the encoding of i using $\log n$ bits even if $\log n$ is not an integer. In general we can imagine encoding a choice from among r options using $\log r$ bits for any $r \in \mathbb{N}$. From this point onwards, we omit ceilings this way in all our theorems and proofs. This simplifies calculations and provides tighter results. For now, it allows us to state the following cleaner version of Theorem 1:

Theorem 1b. *Let $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ be chosen uniformly at random and let $t = \lceil \log n + s \rceil$. Then, the probability that there exists an $i \in \{1, \ldots, n-t+1\}$ such that $x_i = x_{i+1} = \cdots = x_{i+t-1} = 1$ is at most 2^{-s}.*

2.4 Encoding Sparse Bit Strings

At this point we should also point out an extremely useful trick for encoding sparse bit strings. For any $\alpha \in (0,1)$, there exists a code $C_\alpha \colon \{0,1\}^n \to \{0,1\}^*$ such that, for any bit string $x \in \{0,1\}^n$ having $n_1(x)$ ones and $n_0(x)$ zeros,

$$|C_\alpha(x)| = \lceil n_1(x)\log(1/\alpha) + n_0(x)\log(1/(1-\alpha))\rceil \ . \tag{1}$$

This code is the *Shannon-Fano code* for Bernoulli(α) bit strings of length n [15, 36]. More generally, for any probability density $p : X \to [0,1]$, there is a Shannon-Fano code $C : X \to \{0,1\}^*$ such that

$$|C(x)| = \lceil \log(1/p_x)\rceil \ .$$

Moreover, we can construct such a code deterministically, even when X is countably infinite.

Again, as we will see in Section 6, we can omit the ceiling in the expression for $|C_\alpha(x)|$. This is true for any value of n. In particular, for $n = 1$, it gives us a "code" for encoding a single bit where the cost of encoding a 1 is $\log(1/\alpha)$ and the cost of encoding a 0 is $\log(1/(1-\alpha))$. Indeed, the "code" for bit strings of length $n > 1$ is just what we get when we apply this 1-bit code to each bit of the bit string.

If we wish to encode bit strings of length n and we know in advance that the strings contain exactly k one bits, then we can obtain an optimal code by taking $\alpha = k/n$. The resulting fixed length code has length

$$k\log(n/k) + (n-k)\log(n/(n-k)) \ . \tag{2}$$

Equation (2) brings us to our next topic: binary entropy.

2.5 Binary Entropy

The *binary entropy function* $H : (0,1) \to (0,1]$ is defined by

$$H(\alpha) = \alpha\log(1/\alpha) + (1-\alpha)\log(1/(1-\alpha))$$

and it will be quite useful. The binary entropy function and two upper bounds on it that we derive below are illustrated in Figure 3.

We have already encountered a quantity that can be expressed in terms of the binary entropy. From (2), a bit string of length n with exactly k one bits can be encoded with a fixed-length code of $nH(k/n)$ bits.

The binary entropy function can be difficult to work with, so it is helpful to have some manageable approximations. One of these is derived as follows:

$$\begin{aligned} H(\alpha) &= \alpha\log(1/\alpha) + (1-\alpha)\log(1/(1-\alpha)) \\ &= \alpha\log(1/\alpha) + (1-\alpha)\log(1+\alpha/(1-\alpha)) \\ &\le \alpha\log(1/\alpha) + \alpha\log e \end{aligned} \tag{3}$$

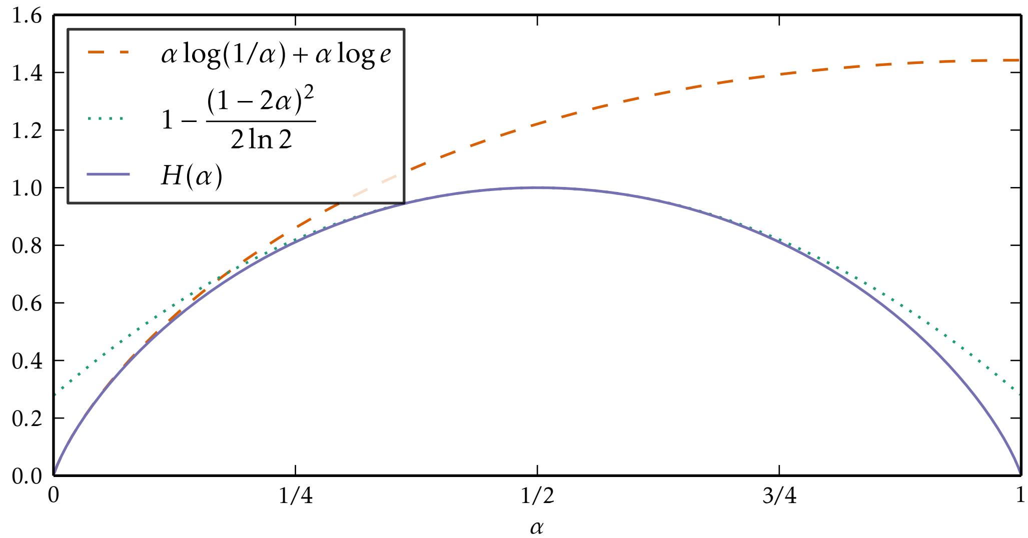


Figure 3: Binary entropy, H, and two useful approximations.

since $1+x \leq e^x$ for all $x \in \mathbb{R}$. Equation (3) is a useful approximation when α is close to zero, in which case $H(\alpha)$ is also close to zero.

For α close to 1/2 (in which case $H(\alpha)$ is close to 1), we obtain a good approximation from the Taylor series expansion at 1/2. Indeed, a simple calculation shows that

$$H'(\alpha) = \log(1/\alpha) - \log(1/(1-\alpha))$$

and that

$$H^{(i)}(\alpha) = \frac{(i-2)!}{\ln 2}\left(\frac{(-1)^{i-1}}{\alpha^{i-1}} - \frac{1}{(1-\alpha)^{i-1}}\right),$$

for $i \geq 2$. Hence, $H^{(i)}(1/2) = 0$, for $i \geq 1$ odd, and

$$H^{(i)}(1/2) = -\frac{(i-2)!\,2^i}{\ln 2},$$

for $i \geq 2$ even. The Taylor series expansion at 1/2 now gives

$$\begin{aligned} H(\alpha) &= H(1/2) + \sum_{i=1}^{\infty} \frac{H^{(i)}(1/2)}{i!}(\alpha - (1/2))^i \\ &= 1 - \frac{1}{\ln 2}\sum_{i=1}^{\infty} \frac{(2i-2)!2^{2i}}{(2i)!2^{2i}}(2\alpha-1)^{2i} \\ &= 1 - \frac{1}{2\ln 2}\sum_{i=1}^{\infty} \frac{(2\alpha-1)^{2i}}{i(2i-1)}. \end{aligned}$$

In particular, for $\alpha = (1-\varepsilon)/2$,

$$H(\alpha) = 1 - \frac{1}{2\ln 2}\sum_{i=1}^{\infty}\frac{\varepsilon^{2i}}{i(2i-1)}$$
$$< 1 - \frac{\varepsilon^2}{2\ln 2} \ . \tag{4}$$

2.6 Basic Chernoff Bound

With all the pieces in place, we can now give an encoding argument for a well-known and extremely useful result typically attributed to Chernoff [8].

Theorem 2. *Let $x \in \{0,1\}^n$ be chosen uniformly at random. Then, for any $\varepsilon \geq 0$,*

$$\Pr\{n_1(x) \leq n(1-\varepsilon)/2\} \leq e^{-\varepsilon^2 n/2} \ .$$

Proof. Encode the bit string x using a Shannon-Fano code C_α with $\alpha = (1-\varepsilon)/2$. Then, the length of the codeword for x is

$$|C_\alpha(x)| = n_1(x)\log(1/\alpha) + n_0(x)\log(1/(1-\alpha)) \ .$$

Since $\alpha < 1/2$, we have $\log(1/\alpha) > \log(1/(1-\alpha))$, so $|C_\alpha(x)|$ is maximal when $n_1(x)$ is maximal. Thus, if $n_1(x) \leq \alpha n$, then

$$\begin{aligned}|C_\alpha(x)| &\leq \alpha n\log(1/\alpha) + (1-\alpha)n\log(1/(1-\alpha)) \\ &= nH(\alpha) \leq n\left(1 - \frac{\varepsilon^2}{2\ln 2}\right) = n - s \ ,\end{aligned}$$

where the second inequality is an application of (4), and where $s = \varepsilon^2 n/(2\ln 2)$. Now, x was chosen uniformly at random from a set of size 2^n. By the Uniform Encoding Lemma, we obtain that

$$\Pr\{n_1(x) \leq \alpha n\} \leq \Pr\{|C_\alpha(x)| \leq n - s\} \leq 2^{-s} = e^{-\varepsilon^2 n/2} \ . \qquad \square$$

In Section 5.1, after developing a Non-Uniform Encoding Lemma, we will extend this argument to Bernoulli(α) bit strings.

2.7 Factorials and Binomial Coefficients

Before moving on to some more advanced encoding arguments, it will be helpful to remind the reader of a few inequalities that can be derived from Stirling's Approximation of the factorial [34]. Recall that Stirling's Approximation states that

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) \ . \tag{5}$$

In many cases, we are interested in representing a set of size $n!$ using a fixed-length code. By (5), and using once again that $1+x \le e^x$ for all $x \in \mathbb{R}$, the length of the codewords in such a code is

$$\begin{aligned}\log n! &= n\log n - n\log e + (1/2)\log n + \log\sqrt{2\pi} + \log(1+\Theta(1/n)) \\ &= n\log n - n\log e + (1/2)\log n + \log\sqrt{2\pi} + \Theta(1/n) \\ &= n\log n - n\log e + \Theta(\log n)\ . \end{aligned} \tag{6}$$

We are sometimes interested in codes for the $\binom{n}{k}$ subsets of k elements from a set of size n. Note that there is an easy bijection between such subsets and binary strings of length n with exactly k ones. Therefore, we can represent these using the Shannon-Fano code $C_{k/n}$ and each of our codewords will have length $nH(k/n)$. In particular, this implies that

$$\log\binom{n}{k} \le nH(k/n) \le k\log n - k\log k + k\log e \tag{7}$$

where the last inequality is an application of (3). The astute reader will notice that we just used an encoding argument to prove an upper-bound on $\binom{n}{k}$ *without knowing the formula* $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Alternatively, we could obtain a slightly worse bound by applying (5) to this formula.

2.8 Encoding the Natural Numbers

So far, we have only explicitly been concerned with codes for finite sets. In this section, we give an outline of some prefix-free codes for the set of natural numbers. Of course, if $p : \mathbb{N} \to [0,1]$ is a probability density, then the Shannon-Fano code for p could serve. However, it seems easier to simply design our codes by hand, rather than find appropriate distributions.

A code is prefix-free if and only if any message consisting of a sequence of its codewords can be decoded unambiguously and instantaneously as it is read from left to right: Consider some sequence of codewords $M = y_1 y_2 \cdots y_k$ from a prefix-free code C. Since C is prefix-free, then C has no codeword z which is a prefix of y_1, so reading M from left to right, the first codeword of C which we recognize is precisely y_1. Continuing in this manner, we can decode the whole message M. Conversely, if for each codeword y of C, a message consisting of this single codeword can be decoded unambiguously and instantaneously from left to right, we know that y has no prefix among the codewords of C, *i.e.* C is prefix-free. This idea allows us to more easily design the codes in this section, which were originally given by Elias [12].

The *unary encoding* of an integer $i \in \mathbb{N}$, denoted by $U(i)$, begins with i 1 bits which are punctuated with a 0 bit. This code is not particularly useful in itself, but it can be improved as follows: The *Elias γ-code* for i, denoted by $E_\gamma(i)$, begins with the unary encoding of the number of bits in i, and then the binary encoding of i itself (minus its leading bit). The *Elias δ-code* for i, denoted by $E_\delta(i)$ begins with an Elias γ-code for the number of bits in i, and then the binary encoding of i itself (minus its leading bit). This process can be continued recursively to obtain the *Elias ω-code*, which we denote by E_ω. Each of these codes has a decoding procedure as in the preceding paragraph, which establishes their prefix-freeness.

The most important properties of these codes are their codeword lengths:

$$\begin{aligned}
|U(i)| &= i+1 \ , \\
|E_\gamma(i)| &= 2\log i + O(1) \ , \\
|E_\delta(i)| &= \log i + 2\log\log i + O(1) \ , \\
|E_\omega(i)| &= \log i + \log\log i + \cdots + \underbrace{\log\cdots\log}_{\log^* i \text{ times}} i + O(\log^* i) \ .
\end{aligned}$$

It may be worth noting that the lengths of unary codes correspond to the lengths of Shannon-Fano codes for a geometric distribution with density $p_i = 1/2^{i+1}$, that is,

$$\log(1/p_i) = \log 2^{i+1} = |U(i)| \ ,$$

and the lengths of Elias γ-codes correspond to the lengths of Shannon-Fano codes for a discrete Cauchy distribution with density $p_i = c/i^2$ for a normalization constant c, that is,

$$\log(1/p_i) = 2\log i - \log c = |E_\gamma(i)| + O(1) \ .$$

The lengths of Elias γ-codes and ω-codes do not seem to arise as the lengths of Shannon-Fano codes for named distributions.

3 Applications of the Uniform Encoding Lemma

We now start with some applications of the Uniform Encoding Lemma. In each case, we will design and analyze a partial prefix-free code $C\colon X \nrightarrow \{0,1\}^*$, where X depends on the context.

3.1 Graphs with no Large Clique or Independent Set

The *Erdős-Rényi random graph $G_{n,p}$* is the probability space on graphs with vertex set $V = \{1,\ldots,n\}$ in which each edge $\{u,w\} \in \binom{V}{2}$ is present with probability p and absent with probability $1-p$, independently of the other edges. Erdős [13] used the random graph $G_{n,\frac{1}{2}}$

to prove that there are graphs that have neither a large clique nor a large independent set. Here we show how this can be done using an encoding argument.

Theorem 3. *For $n \geq 3$ and $s \in \mathbb{N}$, the probability that $G \in G_{n,\frac{1}{2}}$ contains a clique or an independent set of size $t = \lceil 3\log n + \sqrt{2s} \rceil$ is at most 2^{-s}.*

Proof. This is an encoding argument that compresses the $\binom{n}{2}$ bits of G's adjacency matrix, as they appear in row-major order.

Suppose the graph G contains a clique or an independent set S of size t. The encoding $C(G)$ begins with a bit indicating whether S is a clique or independent set; followed by the set of vertices of S; then the adjacency matrix of G in row major-order, omitting the $\binom{t}{2}$ bits implied by the edges or non-edges in S. Such a codeword has length

$$|C(G)| = 1 + t\log n + \binom{n}{2} - \binom{t}{2} \ . \tag{8}$$

Before diving into the detailed arithmetic, we intuitively argue why we're heading in the right direction: Roughly, (8) is of the form:

$$|C(G)| = \binom{n}{2} + t\log n - \Omega(t^2) \ .$$

That is, we need to invest $O(t\log n)$ bits to encode the vertex set of a clique or an independent set of size t, but we save $\Omega(t^2)$ bits in the encoding of G's adjacency matrix. Clearly, for $t > c\log n$, with c sufficiently large, this has the form

$$|C(G)| = \binom{n}{2} - \Omega(t^2) \ .$$

At this point, it is just a matter of pinning down the dependence on c. A detailed calculation beginning from (8) gives

$$\begin{aligned}
|C(G)| &= \binom{n}{2} + 1 + t\log n - (1/2)(t^2 - t) \\
&= \binom{n}{2} + 1 - (1/2)(t^2 - t - 2t\log n) \ .
\end{aligned}$$

The function $f(x) = (1/2)(x^2 - x - 2x\log n) - 1$ is increasing for $x \geq \log n + 1/2$, so recalling that $t = \lceil 3\log n + \sqrt{2s} \rceil$, we get

$$\begin{aligned}
f(t) &\geq f(3\log n + \sqrt{2s}) \\
&= (1/2)(9\log^2 n + 6\sqrt{2s}\log n + 2s - 3\log n - \sqrt{2s} - 6\log^2 n - 2\sqrt{2s}\log n) - 1 \\
&= (1/2)(3\log^2 n + 4\sqrt{2s}\log n - 3\log n - \sqrt{2s}) + s - 1 \\
&\geq s
\end{aligned}$$

for $n \geq 3$. Therefore, our code has length

$$|C(G)| = \binom{n}{2} - f(t)$$
$$\leq \binom{n}{2} - s \ .$$

Applying the Uniform Encoding Lemma completes the proof. □

Remark 1. The bound in Theorem 3 can be strengthened a little, since the elements of S can be encoded using only $\log\binom{n}{t}$ bits, rather than $t \log n$. With a more careful calculation, using (7), the proof then works with $t = 2\log n + O(\log\log n) + \sqrt{s}$. This comes closer to Erdős' original result, which was at the threshold $2\log n - 2\log\log n + O(1)$ [13].

3.2 Balls in Urns

The random experiment of throwing n balls uniformly and independently at random into n urns is a useful abstraction of many questions encountered in algorithm design, data structures, and load-balancing [24, 28]. Here we show how an encoding argument can be used to prove the classic result that, when we do this, no urn contains more than $O(\log n/\log\log n)$ balls.

Theorem 4. *Let $n, s \in \mathbb{N}$, and let t be such that $t\log(t/e) \geq \log n + s$. Suppose we throw n balls independently and uniformly at random into n urns. Then, for sufficiently large n, the probability that any urn contains more than t balls is at most 2^{-s}.*

Before proving Theorem 4, we note that, for any constant $\varepsilon > 0$ and all sufficiently large n, taking

$$t = \left\lceil \frac{(1+\varepsilon)\log n}{\log\log n} \right\rceil$$

satisfies the requirements of Theorem 4, since then

$$t\log t \geq \frac{(1+\varepsilon)\log n}{\log\log n}\log\left(\frac{(1+\varepsilon)\log n}{\log\log n}\right)$$
$$= (1+\varepsilon)\log n\frac{\log\log n}{\log\log n} - (1+\varepsilon)\log n\frac{\log\left(\frac{\log\log n}{1+\varepsilon}\right)}{\log\log n}$$
$$\geq \log n + \varepsilon\log n - (\varepsilon/2)\log n$$
$$= \log n + (\varepsilon/2)\log n$$

for a sufficiently large choice of n. Then,

$$t\log(t/e) = t\log t - t\log e \geq \log n + (\varepsilon/2)\log n - o(\log n) \geq \log n + s \ ,$$

for sufficiently large n, as claimed.

Proof of Theorem 4. For each $i \in \{1,\dots,n\}$, let b_i denote the index of the urn chosen for the i-th ball. The sequence $b = (b_1,\dots,b_n)$ is sampled uniformly at random from a set of size n^n, and this choice will be used in our encoding argument.

Suppose that urn j contains t or more balls. Then, we encode the sequence b with the value j, followed by a code that describes t of the balls in urn j, followed by the remaining $n-t$ values in b that cannot be deduced from the preceding information. Thus, we get

$$\begin{aligned}
|C(b)| &= \log n + \log\binom{n}{t} + (n-t)\log n \\
&= \log n + t\log n - t\log t + t\log e + (n-t)\log n && \text{(using (7))} \\
&= n\log n + \log n - t\log t + t\log e \\
&\le n\log n - s && \text{(by the choice of } t\text{)} \\
&= \log n^n - s
\end{aligned}$$

bits. We conclude the proof by applying the Uniform Encoding Lemma. □

3.3 Linear Probing

Studying balls in urns as in the previous section is useful when analyzing hashing with chaining (see *e.g.* [25, Section 5.1]). A more practically efficient form of hashing is *linear probing*. In a linear probing hash table, we hash the elements of the set $X = \{x_1,\dots,x_n\}$ into a hash table of size $m = cn$, for some fixed $c > 1$. We are given a hash function $h : X \to \{1,\dots,m\}$ which we assume to be a uniform random variable. To insert x_i, we try to place it at table position $j = h(x_i)$. If this position is already occupied by one of $x_1,\dots,x_{i-1}$, we try table location $(j+1) \bmod m$, followed by $(j+2) \bmod m$, and so on, until we find an empty spot for x_i. To find a given element $x \in X$ in the hash table, we compute $j = h(x)$, and we start a linear search from position j until we encounter either x or the first empty position. Assuming that the hash table has been created by inserting the elements from X successively according to the algorithm above, we want to study the expected search time for some item $x \in X$.

We call a maximal consecutive sequence of occupied table locations a *block*. (The table locations $m-1$ and 0 are considered consecutive.)

Theorem 5. *Let $n \in \mathbb{N}$, $c > e$. Suppose that a set $X = \{x_1,\dots,x_n\}$ of n items has been inserted into a hash table of size $m = cn$, using linear probing. Let $t \in \mathbb{N}$ such that*

$$t\log(c/e) - \log t \ge s + O(1) \ ,$$

and fix some $x \in X$. Then the probability that the block containing x has size t is at most 2^{-s}.

Proof. This is an encoding argument for the sequence

$$h = (h(x_1), h(x_2), \ldots, h(x_n)) \ ,$$

that is drawn uniformly at random from a set of size $m^n = (cn)^n$.

Suppose that x lies in a block with t elements. We encode h by the first index b of the block containing x; followed by the $t-1$ elements $y_1, \ldots, y_{t-1}$ of this block (excluding x); followed by information to decode the hash values of x and of $y_1, \ldots, y_{t-1}$; followed by the $n-t$ hash values for the remaining elements in X.

Since the values $h(x), h(y_1), h(y_2), \ldots, h(y_{t-1})$ are in the range $b, \ldots, b+t-1$ (modulo m), they can be encoded using $t \log t$ bits. Therefore, we obtain a codeword of length

$$\begin{aligned}
|C(h)| &= \overbrace{\log m}^{b} + \overbrace{\log\binom{n}{t-1}}^{y_1,\ldots,y_{t-1}} + \overbrace{t\log t}^{h(x),h(y_1),\ldots,h(y_{t-1})} + \overbrace{(n-t)\log m}^{\text{everything else}} \\
&\le \log m + (t-1)\log n - (t-1)\log(t-1) + (t-1)\log e + t\log t + (n-t)\log m \qquad \text{(by (7))} \\
&= (n-t+1)\log m + (t-1)\log(m/c) + (t-1)\log e + \log(t-1) + t\log(t/(t-1)) \quad (m = cn) \\
&= n\log m - (t-1)\log c + (t-1)\log e + \log t + O(1) \\
&= \log m^n - (t-1)\log(c/e) + \log t + O(1) \\
&\le \log m^n - s \ ,
\end{aligned}$$

since we assumed that t satisfies

$$t\log(c/e) - \log t \ge s + O(1) \ .$$

The proof is complete by applying the Uniform Encoding Lemma. $\square$

Remark 2. The proof of Theorem 5 only works if the factor c in the size $m = cn$ of the linear probing hash table is $c > e$. We know from previous analysis that this is not necessary, and that any $c > 1$ is sufficient [35, Theorem 9.8]. We leave it as an open problem to find an encoding proof of Theorem 5 that works for any $c > 1$.

Corollary 1. *Let $n \in \mathbb{N}$, $c > e$. Suppose that a set $X = \{x_1, \ldots, x_n\}$ of n items has been inserted into a hash table of size $m = cn$, using linear probing. Fix some $x \in X$. Then, the expected search time for x in the hash table is $O(1)$.*

Proof. Let T denote the size of the block containing x in the hash table. Let t_0 be a large enough constant. Then, by Theorem 5, the probability that $T = t + t_0$ is at most $2^{-t\log(c/e)/2}$, since then

$$(t+t_0)\log(c/e) - \log(t+t_0) \ge (t+t_0)\log(c/e)/2 \ge t\log(c/e)/2 + O(1) \ .$$

Thus, the expected search time for x is

$$
\begin{aligned}
E\{T\} &= \sum_{t=1}^{\infty} t \Pr\{T = t\} = \sum_{t=1}^{t_0} t \Pr\{T = t\} + \sum_{t=1}^{\infty} (t + t_0) \Pr\{T = t + t_0\} \\
&\le t_0^2 + \sum_{t=1}^{\infty} (t + t_0) 2^{-t \log(c/e)/2} = t_0^2 + \sum_{t=1}^{\infty} (t + t_0)(c/e)^{-t/2} = O(1) \ . \qquad \square
\end{aligned}
$$

3.4 Cuckoo Hashing

Cuckoo hashing is relatively new hashing scheme that offers an alternative to classic perfect hashing [30]. We present a clever proof, due to Mihai Pătraşcu, that cuckoo hashing succeeds with probability $1 - O(1/n)$ [32] (see also [18] for a more detailed exposition of the argument).

We again hash the elements of the set $X = \{x_1, \ldots, x_n\}$. The hash table consists of two arrays A and B, each of size $m = 2n$, and two hash functions $h, g : X \to \{1, \ldots, m\}$ which are uniform random variables. To insert an element x into the hash table, we insert it into $A[h(x)]$; if $A[h(x)]$ already contains an element y, we insert y into $B[g(y)]$; if $B[g(y)]$ already contains some element z, we insert z into $A[h(z)]$, etc. If an empty location is eventually found, the algorithm terminates successfully. If the algorithm runs for too long without successfully completing the insertion, then we say that the insertion failed, and the hash table is rebuilt using different newly sampled hash functions. Any element x either is held in $A[h(x)]$ or $B[g(x)]$, so we can search for x in constant time. The following pseudocode describes this procedure more precisely:

```
Insert(x):
  if x = A[h(x)] or x = B[g(x)] then
    return
  for MaxLoop iterations do
    if A[h(x)] is empty then
      A[h(x)] ← x
      return
    x ↔ A[h(x)]
    if B[g(x)] is empty then
      B[g(x)] ← x
      return
    x ↔ B[g(x)]
  Rehash()
  Insert(x)
```

The threshold 'MaxLoop' is to be specified later. To study the performance of insertion in cuckoo hashing, we consider the random bipartite *cuckoo graph* $G = (A, B, E)$, where $|A| = |B| = m$ and $|E| = n$, with each vertex corresponding either to a location in the array A or B above, and with edge multiset $E = \{(h(x_i), g(x_i)) : 1 \le i \le n\}$.

An *edge-simple* path in G is a path that uses each edge at most once. One can check that if a successful insertion takes at least $2t$ steps, then the cuckoo graph contains an edge simple path with at least t edges. Thus, in bounding the length of edge-simple paths in the cuckoo graph, we bound the worst case insertion time.

Lemma 2. *Let $s \in \mathbb{N}$. Suppose that we insert a set $X = \{x_1, \ldots, x_n\}$ into a hash table using cuckoo hashing. Let G be the resulting cuckoo graph. Then, G has an edge-simple path of length at least $s + \log n + O(1)$ with probability at most 2^{-s}.*

Proof. We encode G by presenting its set of edges. Since each endpoint of an edge is chosen independently and uniformly at random from a set of size m, the set of all edges is chosen uniformly at random from a set of size m^{2n}.

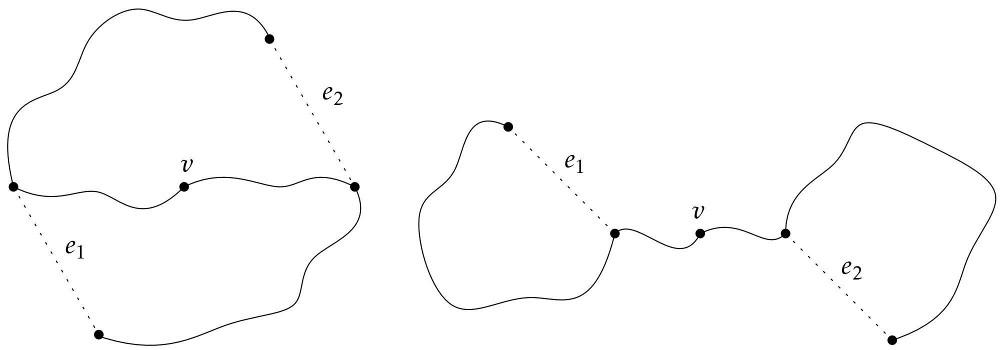
Suppose some vertex $v \in A \cup B$ is the endpoint of an edge-simple path of length t; such a path has $t+1$ vertices and t edges. Each edge in the path corresponds to an element in X. In the encoding, we present the indices of the elements in X corresponding to the t edges of the path in order; then, we indicate whether $v \in A$ or $v \in B$; and we give the $t+1$ vertices in order starting from v; followed by the remaining $2n - 2t$ endpoints of edges of the graph. This code has length

$$\begin{aligned}
|C(G)| &= t \log n + 1 + (t+1) \log m + (2n - 2t) \log m \\
&= 2n \log m + t \log n - t \log m + \log m + O(1) \\
&= \log m^{2n} - t + \log n + O(1) && \text{(since } m = 2n\text{)} \\
&\le \log m^{2n} - s
\end{aligned}$$

for $t \ge s + \log n + O(1)$. We finish by applying the Uniform Encoding Lemma. □

This immediately implies that a successful insertion takes time at most $4 \log n + O(1)$ with probability $1 - O(1/n)$. Moreover, selecting 'MaxLoop' to be $4 \log n + O(1)$, we see that a rehash happens only with probability $O(1/n)$.

One can prove that the cuckoo hashing insertion algorithm fails if and only if some subgraph of the cuckoo graph contains more edges than vertices, since edges correspond to keys, and vertices correspond to array locations.



(a) A cycle with a chord. (b) Two cycles connected by a path.

Figure 4: The potential minimal subgraphs of the cuckoo graph.

Lemma 3. *The cuckoo graph has a subgraph with more edges than vertices with probability $O(1/n)$. In other words, cuckoo hashing insertion succeeds with probability $1 - O(1/n)$.*

Proof. Suppose that some vertex v is part of a subgraph with more edges than vertices, and in particular a minimal such subgraph with $t+1$ edges and t vertices. Such a subgraph appears exactly as in Figure 4. By inspection, we see that for every such subgraph, there are two edges e_1 and e_2 whose removal disconnects the graph into two paths of length t_1 and t_2 starting from v, where $t_1 + t_2 = t - 1$.

We encode G by giving the vertex v; and presenting Elias δ-codes for the values of t_1 and t_2 and for the positions of the endpoints of e_1 and e_2; then the indices of the edges of the above paths in order; then the vertices of the paths in order; and the indices of the edges e_1 and e_2; and finally the remaining $2n - 2(t+1)$ endpoints of edges in the graph. Such a code has length

$$\begin{aligned}
|C(G)| &= \log m + O(\log t) + (t-1)(\log n + \log m) + 2\log n + (2n - 2(t+1))\log m \\
&= 2n\log m + (t+1)\log n - (t+2)\log m + O(\log t) \\
&= 2n\log m + (t+1)\log n - (t+2)\log n - t + O(\log t) \qquad \text{(since } m = 2n\text{)} \\
&\le \log m^{2n} - \log n + O(1)\ .
\end{aligned}$$

We finish by applying the Uniform Encoding Lemma. □

3.5 2-Choice Hashing

We showed in Section 3.2 that if n balls are thrown independently and uniformly at random into n urns, then the maximum number of balls in any urn is $O(\log n/\log\log n)$ with high probability. In *2-choice hashing*, each ball is instead given a choice of two urns, and the urn containing the fewer balls is preferred.

More specifically, we are given two hash functions $h, g : X \to \{1, \ldots, m\}$ which are uniform random variables. Each value of h and g points to one of m urns. The element $x \in X$ is added to the urn containing the fewest elements between $h(x)$ and $g(x)$ during an insertion. The worst case search time is at most the maximum number of hashing collisions, or the maximum number of elements in any urn.

Perhaps surprisingly, the simple change of having two choices instead of only one results in an exponential improvement over the strategy of Section 3.2. The concept of 2-choice hashing was first studied by Azar *et al.* [2], who showed that the expected maximum size of an urn is $\log\log n + O(1)$. Our encoding argument is based on Vöcking's use of witness trees to analyze 2-choice hashing [38].

Let $G = (V, E)$ be the random multigraph with $V = \{1, \ldots, m\}$, where $m = cn$ for some constant $c > 8$, and $E = \{(h(x), g(x)) : x \in X\}$. Each edge in E is labeled with the element $x \in X$ that it corresponds to.

Lemma 4. *The probability that G has a subgraph with more edges than vertices is $O(1/n)$.*

Proof. The proof is similar to that of Lemma 3. More specifically, we can encode G by giving the same encoding as in Lemma 3, with an additional bit for each edge uv in the encoding, indicating whether $u = h(x)$ and $v = g(x)$, or $u = g(x)$ and $v = h(x)$. Our code thus has length

$$\begin{aligned} |C(G)| &= \log m + (t-1)(\log n + \log m) + 2\log n + (2n - 2(t+1))\log m + t + O(\log t) \\ &= 2n\log m - \log n - t\log c + t + O(\log t) \\ &\le \log m^{2n} - \log n + O(1) \ , \end{aligned}$$

since $\log c > 1$. □

Lemma 5. *G has a component of size at least $(2/\log(c/8))\log n + O(1)$ with probability $O(1/n)$.*

Proof. Suppose G has a connected component with t vertices and at least $t-1$ edges. This component has a spanning tree T. Pick an arbitrary vertex as the root of T. To encode G, we specify a bit string encoding the shape of T, then the t vertices and the elements in X corresponding to the $t-1$ edges encountered in a pre-order traversal of T; and finally the remaining $2(n-t+1)$ endpoints of edges in G. For each edge uv in T we also store a bit indicating whether $u = h(x)$ and $v = g(x)$, or $u = g(x)$ and $v = h(x)$.

We can encode the shape of T with a bit string of length $2(t-1)$, tracing a pre-order traversal of the tree, where a 0 bit indicates that the path to the next node goes up, and a 1 bit indicates that the path goes down. In total, our code has length

$$\begin{aligned}
|C(G)| &= t\log m + (t-1)(\log n + 1) + 2(t-1) + 2(n-t+1)\log m \\
&= 2n\log m + t\log c - \log n + 3t - 3 - 2t\log c + 2\log n + 2\log c \qquad \text{(since } m = cn\text{)} \\
&= \log m^{2n} - t\log c + 3t + \log n + O(1) \\
&\le \log m^{2n} - s \ ,
\end{aligned}$$

as long as t is such that

$$t \ge \frac{s + \log n + O(1)}{\log(c/8)} \ .$$

In particular, for $s = \log n$, the Uniform Encoding Lemma tells us that G has a component of size at least $(2/\log(c/8))\log n + O(1)$ with probability $O(1/n)$. □

Suppose that when x is inserted, it is placed in an urn with t other elements. Then, we say that the *age* of x is $a(x) = t$.

Theorem 6. *Fix $c > 8$ and suppose we insert n elements into a table of size cn using 2-choice hashing. With probability $1 - O(1/n)$ all positions in the hash table contain at most $\log\log n + O(1)$ elements.*

Proof. Suppose that some element x has $a(x) = t$. This leads to a binary *witness tree* T of height t as follows: The root of T is the element x. When x was inserted into the hash table, it had to choose between the urns $h(x)$ and $g(x)$, both of which contained at least $t-1$ elements; in particular, $h(x)$ has a unique element x_h with $a(x_h) = t-1$, and $g(x)$ has a unique element x_g with $a(x_g) = t-1$. The elements x_h and x_g become the left and the right child of x in T. The process continues recursively. If some element appears more than once on a level, we only recurse on its leftmost occurrence. See Figure 5 for an example.

Using T, we can iteratively define a connected subgraph G_T of G. Initially, G_T consists of the single node in V corresponding to the bucket that contains the root element x of T. Now, to construct G_T, we go through T level by level, starting from the root. For $i = t, \dots, 0$, let L_i be all elements in T with age i, and let $E_i = \{(h(x), g(x)) : x \in L_i\}$ be the corresponding edges in G. When considering L_i, we add to G_T all edges in E_i, together with their endpoints, if they are not in G_T already. Since every element appears at most once in T, this adds $|L_i|$ new edges to G_T. The number of vertices in G_T increases by at most $|L_i|$. In the end, G_T contains $\sum_{i=0}^{t} |L_i|$ edges. Since G_T is connected, with probability $1 - O(1/n)$, the number of edges in G_T does not exceed the number of vertices, by Lemma 4. We

	h	g
$\vdots$	$\vdots$	$\vdots$
x_5	1	1
x_6	2	3
x_7	4	2
x_8	1	2
x_9	1	4
x_{10}	1	4

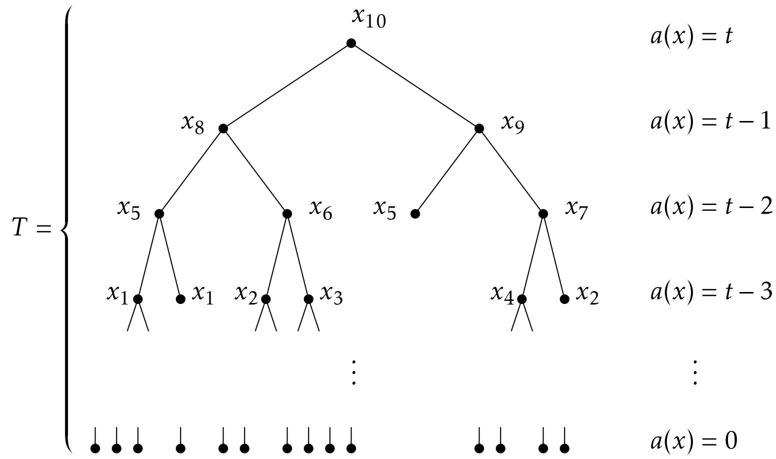


Figure 5: The tree T is a witness tree for the 2-choice hashing instance with elements $\ldots, x_5, x_6, x_7, x_8, x_9, x_{10}$ inserted in order and according to the hash functions h and g.

assume that this is the case. Since initially G_T had one vertex and zero edges, the iterative procedure must add at least $\sum_{i=0}^{t} |L_i| - 1$ new vertices to G_T. This means that all nodes in T but one must have two children, so we can conclude that T is a complete binary tree with at most one subtree removed. It follows that T (and hence G_T) has at least 2^t vertices. If we choose $t = \lceil \log\log n + d \rceil$, then $2^t \geq 2^d \log n$. We know from Lemma 5 that this happens with probability $O(1/n)$ for a sufficiently large choice of the constant d. □

Remark 3. The arguments in this section can be refined by more carefully encoding the shape of trees using *Cayley's formula*, which says that there are t^{t-2} unrooted labelled trees on t nodes [7]. In particular, an unrooted tree with t nodes and m choices for distinct node labels can be encoded using $\log\binom{m}{t} + (t-2)\log t$ bits instead of $t \log m + 2(t-1)$ bits. We would then recover the same results for hash tables of size $m = cn$ with $c > 2e$ instead of $c > 8$. In fact, it is known that for *any* $c > 0$ searching in 2-choice hashing takes time $1/c + O(\log\log n)$ [5]. We leave it as an open problem to find an encoding argument for this result when $c > 0$.

Remark 4. *Robin Hood hashing* is another hashing solution which achieves $O(\log\log n)$ worst case running time for all operations [10]. The original analysis is difficult, but might be amenable to a similar approach as we used in this section. Indeed, when a Robin Hood hashing operation takes a significant amount of time, a large witness tree is again implied, which suggests an easy encoding argument. Unfortunately, this approach appears to involve unwieldy hypergraph encoding.

3.6 Bipartite Expanders

Expanders are families of sparse graphs which share many connectivity properties with the complete graph. These graphs have received much research attention, and have led to many applications in computer science. See, for instance, the survey by Hoory, Linial, and Wigderson [20].

The existence of expanders was originally established through probabilistic arguments [31]. We offer an encoding argument to prove that a certain random bipartite graph is an expander with high probability. There are many different notions of expansion. We will consider what is commonly known as *vertex expansion* in bipartite graphs: For some fixed $0 < \alpha \leq 1$, a bipartite graph $G = (A, B, E)$ is called a (c, α)-*expander* if

$$\min_{\substack{A' \subseteq A \\ |A'| \leq \alpha |A|}} \frac{|N(A')|}{|A'|} \geq c \ ,$$

where $N(A') \subseteq B$ is the set of neighbours of A' in G. That is, in a (c, α)-expander, every set of vertices in A that is not too large is "expanded" by a factor c by taking one step in the graph.

Let $G = (A, B, E)$ be a random bipartite multigraph where $|A| = |B| = n$ and where each vertex of A is connected to three vertices of B chosen independently and uniformly at random (with replacement). The following theorem shows that G is an expander. The proof of this theorem usually involves a messy sum that contains binomial coefficients and probabilities: see, for example, Motwani and Raghavan [28, Theorem 5.3], Pinsker [31, Lemma 1], or Hoory, Linial, and Wigderson [20, Lemma 1.9].

Theorem 7. *There exists a constant* $\alpha > 0$ *such that* G *is a* $(3/2, \alpha)$*-expander with probability at least* $1 - O(n^{-1/2})$.

Proof. We encode the graph G by presenting its edge set. Since each edge is selected uniformly at random, the graph G is chosen uniformly at random from a set of size n^{3n}.

If G is not a $(3/2, \alpha)$-expander, then there is some set $A' \subseteq A$ with $|A'| = k \leq \alpha n$ and

$$\frac{|N(A')|}{|A'|} < 3/2 \ .$$

To encode G, we first give k using an Elias γ-code; together with the sets A' and $N(A')$; and the edges between A' and $N(A')$. Then we encode the rest of G, skipping the $3k \log n$ bits devoted to edges incident to A'. The key savings here come because $N(A')$ should take

$3k\log n$ bits to encode, but can actually be encoded in roughly $3k\log(3k/2)$ bits. Our code then has length

$$\begin{aligned}|C(G)| &= 2\log k + \log\binom{n}{k} + \log\binom{n}{3k/2} + 3k\log(3k/2) + (3n-3k)\log n + O(1) \\ &\le 2\log k + k\log n - k\log k + k\log e + (3k/2)\log n - (3k/2)\log(3k/2) \\ &\qquad + (3k/2)\log e + 3k\log(3k/2) + (3n-3k)\log n + O(1) \qquad \text{(by (7))} \\ &= 3n\log n - (k/2)\log n + (k/2)\log k + \beta k + 2\log k + O(1) \\ &= \log n^{3n} - s(k)\end{aligned}$$

bits, where $\beta = (3/2)\log(3/2) + (5/2)\log e$ and

$$s(k) = (k/2)\log n - (k/2)\log k - \beta k - 2\log k - O(1) \ .$$

Since

$$\frac{d^2}{dk^2}s(k) = \frac{4-k}{2k^2}\log e \ ,$$

the function $s(k)$ is concave for all $k \ge 4$. Thus, $s(k)$ is minimized either when $k = 1, 2, 3, 4$, or when $k = \alpha n$. We have

$$\begin{aligned} s(1) &= (1/2)\log n + c_1 \ , & s(2) &= \log n + c_2 \ , \\ s(3) &= (3/2)\log n + c_3 \ , & s(4) &= 2\log n + c_4 \ , \end{aligned}$$

for constants c_1, c_2, c_3, c_4. For $k = \alpha n$ we have

$$s(\alpha n) = (\alpha n/2)\log\left(\frac{1}{2^{2\beta}\alpha}\right) - 2\log \alpha n + c_5,$$

for some constant c_5. Thus, $2^{-s(\alpha n)} = 2^{-\Omega(n)}$ for $\alpha < (1/2)^{2\beta} \approx 0.002$. Now the Uniform Encoding Lemma gives the desired result. Indeed, the encoding works for all values of k, and it always saves at least $s(1) = (1/2)\log n + O(1)$ bits. Thus, the probability that the construction fails is at most $O(n^{-1/2})$. □

3.7 Permutations and Binary Search Trees

We define a *permutation* σ of size n to be a sequence of n pairwise distinct integers, sometimes denoted by $\sigma = (\sigma_1, \ldots, \sigma_n)$. The set $\{\sigma_1, \ldots, \sigma_n\}$ is called the *support* of σ. This slightly unusual definition will serve us for the purpose of encoding. Except when explicitly stated, we will assume that the support of a permutation of size n is precisely $\{1, \ldots, n\}$. For any fixed support of size n, the number of distinct permutations is $n!$.

3.7.1 Analysis of Insertion Sort

Recall the insertion-sort algorithm for sorting a list $\sigma = (\sigma_1, \ldots, \sigma_n)$ of n elements:

```
InsertionSort(σ)
1: for i ← 2 to n do
2:   j ← i
3:   while j > 1 and σ_{j−1} > σ_j do
4:     σ_j ↔ σ_{j−1} { swap }
5:     j ← j − 1
```

A typical task in the average-case analysis of algorithms is to determine the number of times Line 4 executes if σ is a uniformly random permutation of size n. The answer $\binom{n}{2}/2$ is an easy application of linearity of expectation: For every one of the $\binom{n}{2}$ pairs of indices $p, q \in \{1, \ldots, n\}$ with $p < q$, the values initially stored at positions σ_p and σ_q will eventually be swapped if and only if $\sigma_p > \sigma_q$. This happens with probability 1/2 in a uniformly random permutation. A pair $p, q \in \{1, \ldots, n\}$ with $p < q$ and $\sigma_p > \sigma_q$ is called an *inversion*, so the number of times Line 4 executes is the number of inversions of σ.

A more advanced question is to ask for a concentration result on the number of inversions. This is harder to tackle; because $>$ is transitive, the $\binom{n}{2}$ events being studied have a lot of interdependence. In the following, we show how an encoding argument leads to a concentration result. The argument presented here follows the same outline as Vitányi's analysis of bubble sort [37], though without all the trappings of Kolmogorov complexity.

Theorem 8. *Let $\alpha \in (0, 1/e^2)$. A uniformly random permutation σ of size n has at most $\alpha n^2 - n + 2$ inversions with probability at most $2^{n\log(\alpha e^2)+O(\log n)}$. In particular, for a fixed $\alpha < 1/e^2$, this probability is $2^{-\Omega(n)}$.*

Proof. We encode the permutation σ by recording the execution of InsertionSort on σ. In particular, we record for each $i \in \{2, \ldots, n\}$, the number of times m_i that Line 4 executes during the i-th iteration of InsertionSort(σ). With this information, one can run the following algorithm to recover σ:

```
InsertionSortReconstruct(m_2, ..., m_n):
1: σ ← (1, ..., n)
2: for i ← n down to 2 do
3:   for j ← i − m_i + 1 to i do
4:     σ_j ↔ σ_{j−1} { swap }
```

5: **return** σ

We have to be slightly clever with the encoding. Rather than encode $m_2, m_3, \dots, m_n$ directly, we first encode $m = \sum_{i=2}^{n} m_i$ using $2\log n$ bits (since $m < n^2$). Given m, it remains to describe the partition of m into $n-1$ non-negative integers $m_2, \dots, m_n$; there are $\binom{m+n-2}{n-2}$ such partitions.[2]

Therefore, the values of $m_2, \dots, m_n$ can be encoded using

$$|C(\sigma)| = 2\log n + \log\binom{m+n-2}{n-2}$$

bits and this is sufficient to recover the permutation σ. By applying (7), we obtain

$$\begin{aligned}
|C(\sigma)| &\le (n-2)\log(m+n-2) - (n-2)\log(n-2) + (n-2)\log e + O(\log n) \\
&\le n\log(m+n-2) - n\log n + n\log e + O(\log n) \\
&\le n\log(\alpha n^2) - n\log n + n\log e + O(\log n) && (\text{since } m \le \alpha n^2 - n + 2) \\
&= 2n\log n + n\log\alpha - n\log n + n\log e + O(\log n) \\
&= n\log n + n\log\alpha + n\log e + O(\log n) \\
&= \log n! + n\log\alpha + 2n\log e + O(\log n) && (\text{by } (6)) \\
&= \log n! + n\log(\alpha e^2) + O(\log n) \ .
\end{aligned}$$

Again, we finish by applying the Uniform Encoding Lemma. □

Remark 5. Theorem 8 is not sharp; it only gives a non-trivial probability when $\alpha < 1/e^2$. To obtain a sharp bound, one can use the fact that $m_2, \dots, m_n$ are independent and that m_i is uniform over $\{0, \dots, i-1\}$ together with the method of bounded differences [22]. This shows that m is concentrated in an interval of size $O(n^{3/2})$.

3.7.2 Records

A *(max) record* in a permutation σ of size n is some value σ_i, $1 \le i \le n$, such that

$$\sigma_i = \max\{\sigma_1, \dots, \sigma_i\} \ .$$

If σ is chosen uniformly at random, the probability that σ_i is a record is exactly $1/i$. Thus, the expected number of records in such a permutation is

$$H_n = \sum_{i=1}^{n} 1/i = \ln n + O(1) \ ,$$

[2] To see this, draw $m+n-2$ white dots on a line, then choose $n-2$ dots to colour black. This splits the remaining m white dots into $n-1$ groups whose sizes determine the values of $m_2, \dots, m_n$.

the n-th harmonic number. It is harder to establish concentration with non-negligible probability. To do this, one first needs to show the independence of certain random variables, which quickly becomes tedious. We instead give an encoding argument to show concentration of the number of records, inspired by a technique used by Lucier *et al.* to study the height of random binary search trees [21] (see also Section 3.7.3).

First, we describe a recursive encoding of a permutation σ of size n: Begin by providing the first value of the permutation σ_1; then show the set of indices from $\{2,\ldots,n\}$ for which σ takes on a value strictly smaller than σ_1 and an explicit encoding of the induced permutation on the elements at those indices; finally, give a recursive encoding of the permutation induced on the elements strictly larger than σ_1. The number of recursive invocations is equal to the number of records in σ.

If σ contains k elements strictly smaller than σ_1, then the length $\ell(\sigma)$ of the codeword for σ satisfies

$$\ell(\sigma) = \log n + \log\binom{n-1}{k} + \log k! + \ell(\sigma') \ ,$$

where σ' is the induced permutation on the $n-k-1$ elements strictly larger than σ_1. Thus, we get the following recursion for the length $\ell(n)$ of the encoding for a permutation of size n:

$$\ell(n) = \max_{k\in\{1,\ldots,n-1\}}\left(\log n + \log\binom{n-1}{k} + \log k! + \ell(n-1-k)\right) \ ,$$

with $\ell(0) = 0$ and $\ell(1) = 0$. This solves to $\ell(n) = \log n!$, so the encoding described above is no better than a fixed-length encoding for σ. However, a simple modification of the scheme yields a result about the concentration of records in a uniformly random permutation.

Theorem 9. *For any fixed $c > 2$, a uniformly random permutation σ of size n has at least $c\log n$ records with probability at most*

$$2^{-c(1-H(1/c))\log n+O(\log\log n)} \ .$$

Proof. We describe an encoding scheme for permutations with at least $t = \lceil c\log n\rceil$ records. Suppose that the permutation σ has t records $r_1 < r_2 < \cdots < r_t$. First, we define a bit string $x = (x_1,\ldots,x_t) \in \{0,1\}^t$, where $x_1 = 0$ and $x_i = 1$ if and only if r_i lies in the second half of the interval $[r_{i-1}, n]$, for $i = 2,\ldots,t$. Recalling that $n_1(x)$ represents the number of ones in the bit string x, it follows that $n_1(x) \le \log n$, so $n_1(x)/t \le 1/c$.

To begin our encoding of σ, we encode the bit string x by giving the set of $n_1(x)$ ones in x; followed by the recursive encoding of σ from earlier. Now, our knowledge of the value of x_i halves the size of the space of options for encoding the position r_i . In other words, our knowledge of x allows us to encode each record using roughly one less bit per record. More precisely, if the number of choices for each record r_i in the original encoding is m_i, such that $m_1 > \cdots > m_t$, then the number of bits spent encoding records in the new code is at most

$$\begin{aligned}
\sum_{i=1}^{t} \log\lceil m_i/2\rceil &\le \sum_{i=1}^{t} \log(m_i/2+1) \\
&\le \sum_{i=1}^{t} \log(m_i/2) + \sum_{i=1}^{t} O(1/m_i) && \text{(since } \log(x+1) = \log x + O(1/x)\text{)} \\
&\le \sum_{i=1}^{t} \log(m_i/2) + O(H_t) \\
&= \sum_{i=1}^{t} \log m_i - t + O(\log\log n) \ ,
\end{aligned}$$

since c is a constant. Thus, the total length of the code is

$$\begin{aligned}
|C(\sigma)| &\le \binom{t}{n_1(x)} + \log n! - t + O(\log\log n) \\
&\le \log n! - t(1 - H(n_1(x)/t)) + O(\log\log n) && \text{(by (7))} \\
&\le \log n! - c(1 - H(1/c))\log n + O(\log\log n) \ ,
\end{aligned}$$

where this last inequality follows since $c > 2$, so $0 \le n_1(x)/t \le 1/c < 1/2$, and $H(n_1(x)/t) \le H(1/c)$ since $H(\cdot)$ is increasing on $[0, 1/2]$. We finish by applying the Uniform Encoding Lemma. □

Remark 6. The preceding result only works for $c > 2$, but it is known that the number of records in a uniformly random permutation is concentrated around $\ln n + O(1)$, where $\ln n = \alpha \log n$ for $\alpha = 0.6931\ldots$. We leave as an open problem whether or not this significant gap can be closed through an encoding argument.

3.7.3 The Height of a Random Binary Search Tree

Every permutation σ determines a binary search tree $\mathrm{BST}(\sigma)$ created through the sequential insertion of the keys $\sigma_1, \ldots, \sigma_n$. Specifically, if σ^L (respectively, σ^R) denotes the permutation of elements strictly smaller (respectively, strictly larger) than σ_1, then $\mathrm{BST}(\sigma)$ has σ_1 as its root, with $\mathrm{BST}(\sigma^L)$ and $\mathrm{BST}(\sigma^R)$ as left and right subtrees.

Lucier *et al.* [21] use an encoding argument via Kolmogorov complexity to study the height of $\mathrm{BST}(\sigma)$. They show that for a uniformly chosen permutation σ, the tree $\mathrm{BST}(\sigma)$ has height at most $c \log n$ with probability $1 - O(1/n)$ for $c = 15.498\ldots$; we can extend our result on records from Section 3.7.2 to obtain a tighter result.

For a node u, let $s(u)$ denote the number of nodes in the tree rooted at u. Then, u is called *balanced* if $s(u^L), s(u^R) > s(u)/4$, where u^L and u^R are the left and right subtrees of u, respectively. In other words, since each node u determines an interval $[v, w]$, where v is the smallest node in the subtree rooted at u, and w is the largest such node, then u is balanced if and only if

$$u \in \left(\frac{w+v}{2} - \frac{w-v-1}{4}, \frac{w+v}{2} + \frac{w-v-1}{4}\right) ,$$

i.e. u is called balanced if it occurs inside the middle interval of length $(w-v-1)/2$ of its subrange.

Theorem 10. *Let σ be a uniformly random permutation of size n. There is a constant $c <$ 9.943483 such that $\mathrm{BST}(\sigma)$ has height at most $c \log n$ with probability $1 - O(1/n)$.*

Proof. Let $c > 2/\log(4/3)$, and suppose that the tree $\mathrm{BST}(\sigma)$ contains a path $Y = (y_1, \ldots, y_t)$ of length $t = \lceil c \log n \rceil$ that starts at the root and in which y_{i+1} is a child of y_i, for $i = 1, \ldots, t-1$.

Our encoding for σ has three parts. The first part consists of a bit string $x = (x_1, \ldots, x_t)$, where $x_i = 1$ if and only if y_i is balanced. From our definition, if y_i is balanced, then $s(y_{i+1}) \le (3/4)s(y_i)$. Since $n_1(x)$ counts the number of balanced nodes along Y, we get

$$1 \le (3/4)^{n_1(x)} n \iff n_1(x) \le \log_{4/3} n \ .$$

Next, our encoding contains a fixed-length encoding of y_t using $\log n$ bits.

The third part of our encoding is recursive: First, encode the value of the root y_1 using $\log\lceil n/2 \rceil$ bits. Note that since we know whether y_1 is balanced or not, there are only $n/2$ possibilities for the root value, by the discussion above. If y_2 is the left child of y_1, then specify the values in the right subtree of y_1, including an explicit encoding of the permutation induced by these values; and recursively encode the permutation of values in subtree of y_2. If, instead, y_2 is the right child of y_1, proceed symmetrically. (Note that a decoder can determine which of these two cases occured by comparing y_t with y_1 since $y_2 < y_1$ if and only if $y_t < y_1$.) Once we reach y_t, we encode the permutations of the two subtrees of y_t explicitly.

The first two parts of our encoding use at most

$$tH(n_1(x)/t) + \log n$$

bits. The same analysis as in the proof of Theorem 9 shows that the second part of our encoding has length at most

$$\log n! - t + O(\log\log n) \ .$$

In total, our code has length

$$
\begin{aligned}
|C(\sigma)| &= \log n! - t + tH(n_1(x)/t) + \log n + O(\log\log n) \\
&\le \log n! - c\log n + c\log n H\left(\frac{1}{c\log(4/3)}\right) + \log n + O(\log\log n) \\
&= \log n! - c\left(1 - H\left(\frac{1}{c\log(4/3)}\right)\right) + \log n + O(\log\log n) \ ,
\end{aligned}
$$

where the inequality uses the fact that $c > 2/\log(4/3)$. Applying the Uniform Encoding Lemma, we see that BST(σ) has height at most $c\log n$ with probability $1 - O(1/n)$ for $c > 2/\log(4/3)$ satisfying

$$
c\left(1 - H\left(\frac{1}{c\log(4/3)}\right)\right) > 2 \ ,
$$

and a computer-aided calculation shows that $c = 9.943483$ satisfies this inequality. □

Remark 7. Devroye [9] shows how the length of the path to the key i in BST(σ) relates to the number of records in σ. Specifically, he notes that the number of records in σ is the number of nodes along the rightmost path in BST(σ). Since the height of a tree is the length of its longest root-to-leaf path, we obtain as a corollary that the number of records in a uniformly random permutation is $O(\log n)$ with high probability; the result from Theorem 9 only improves upon the implied constant.

Remark 8. We know that the height of the binary search tree built from the sequential insertion of elements from a uniformly random permutation of size n is concentrated around $\alpha \ln n + O(\log\log n)$, for $\alpha = 4.311\ldots$ [33]. Perhaps if the gap in our analysis of records in Remark 6 can be closed through an encoding argument, then so too can the gap in our analysis of random binary search tree height.

3.7.4 Hoare's Find Algorithm

In this section, we analyze the number of comparisons made in an execution of Hoare's classic Find algorithm [19] which returns the k-th smallest element in an array of n elements. The analysis is similar to that of the preceding section.

We refer to an easy algorithm Partition, which takes as input an array $\sigma = (\sigma_1, \ldots, \sigma_n)$ and partitions it into the arrays σ^L and σ^R which contain the values strictly smaller and strictly larger than σ_1, respectively. The element σ_1 is called a *pivot*. The algorithm Partition can be implemented so as to perform only $n - 1$ comparisons as follows:

Partition(σ):

1: $\sigma^L, \sigma^R \leftarrow$ **nil**

2: **for** $i \leftarrow 2$ **to** n **do**
3:   **if** $\sigma_i > \sigma_1$ **then**
4:     push σ_i onto σ^R
5:   **else**
6:     push σ_i onto σ^L
7: **return** σ^L, σ^R

Using this, we give the algorithm Find:

Find(k, σ):
1: $\sigma^L, \sigma^R \leftarrow$ Partition(σ)
2: **if** $|\sigma^L| \geq k$ **then**
3:   **return** Find(k, σ^L)
4: **else if** $|\sigma^L| < k-1$ **then**
5:   **return** Find$(k - |\sigma^L| - 1, \sigma^R)$
6: **return** σ_1

Suppose that the algorithm Find sequentially identifies t pivots $x_1, \ldots, x_t$ before finding the solution. Let $\sigma^{(i)}$ denote the value of σ in the ith recursive call and let $n_i = |\sigma^{(i)}|$, so that $\sigma^{(0)} = (\sigma_1, \ldots, \sigma_n)$ and $n_0 = n$. We will say that the ith pivot is *good* if its rank, in $\sigma^{(i)}$, is in the interval $[n_i/4, 3n_i/4]$. Note that a good pivot causes the algorithm to recurse in a problem of size at most $3n_i/4$.

Lemma 6. *Fix some constants $t_0 \geq 1$ and $0 < \alpha < 1/2$. Suppose that, for each $t_0 \leq i \leq t$, the number of good pivots among $x_1, \ldots, x_i$ is at least αi. Then,* Find *makes $O(n)$ comparisons.*

Proof. If x_j is a good pivot, then the conditions of the lemma give that $n_j \leq (3/4)n_{j-1}$. Therefore,

$$n_i \leq (3/4)^{\alpha i} n$$

for each $t_0 \leq i \leq t$, and the total number of comparisons made by Find is at most

$$\sum_{i=0}^{t} n_i \leq t_0 n + \sum_{i=t_0}^{t} n_i \leq O(n) + n \sum_{i=t_0}^{t} (3/4)^{\alpha i} = O(n) \ . \qquad \square$$

Theorem 11. *Let σ be a uniformly random permutation. Then, for every fixed probability $p \in (0,1)$, there exists a constant c such that* Find(k, σ) *executes at most cn comparisons with probability at least p, for any k.*

Proof. We again encode the permutation σ. Set $\alpha = 1/4$ and let t_0 be a constant depending on p. Suppose that the conditions of the preceding lemma are not satisfied for α and t_0, *i.e.* there is an $i \geq t_0$ such that the number of good pivots among $x_1, \ldots, x_i$ is less than αi. We encode σ in two parts. The first part of our encoding gives the value of i using an Elias δ-code, followed by the set of indices of the good pivots among $x_1, \ldots, x_i$, which costs

$$\log i + iH(\alpha) + O(\log\log i) \enspace .$$

Note that the pivots $x_1, \ldots, x_i$ trace a path from the root in $\mathrm{BST}(\sigma)$. Therefore, the second part of our encoding is the recursive encoding presented in Section 3.7.3, in which each pivot can be encoded using one less bit, since knowng whether x_j is a good pivot or not narrows down the range of possible values for x_j by 1/2. In total, our code then has length

$$|C(\sigma)| \leq \log n! - i + iH(\alpha) + \log i + O(\log\log i) = \log n! - \Omega(i) \enspace ,$$

since $\alpha = 1/4 < 1/2$. The proof is completed by applying the Uniform Encoding Lemma, and by observing that $t_0 \leq i$ can be made arbitrarily large. □

3.8 k-SAT and the Lovász Local Lemma

We now consider the question of satisfiability of propositional formulas. Let us start with some definitions.

A (Boolean) *variable* x is either true or false. The negation of x is denoted by $\neg x$. A *literal* is either a variable or its negation. A *conjunction* of literals is an "and" of literals, denoted by $\wedge$. A *disjunction* of literals is an "or" of literals, denoted by $\vee$. A *formula* φ is an expression including conjunctions and disjunctions of literals, and the set of variables involved in this formula is called the *support* of φ. A *clause* is a disjunction of literals, *i.e.* the "or" of a set of variables or their negations, *e.g.*

$$x_1 \vee \neg x_2 \vee x_3 \enspace . \tag{9}$$

Two clauses will be said to intersect if their supports intersect. The truth value which a formula φ evaluates to under the assignment of values α to its support will be denoted by $\varphi(\alpha)$, and such a formula is said to be *satisfiable* if there exists an α with $\varphi(\alpha) = \text{true}$. For example, the clause in (9) is satisfied for all truth assignments except

$$(x_1, x_2, x_3) = (\text{false}, \text{true}, \text{false}) \enspace ,$$

and indeed any clause is satisfied by all but one truth assignments for its support. The formulas we are concerned with are conjunctions of clauses, which are said to be in *conjunctive normal form* (CNF). More specifically, when each clause has at most k literals, we call it a k-CNF formula.

The k-SAT decision problem asks to determine whether or not a given k-CNF formula is satisfiable. In general, this problem is hard. Of course, any satisfying truth assignment to the variables in a CNF formula induces a satisfying truth assignment for each of its clauses. Moreover, if the supports of the clauses are pairwise disjoint, then the formula is trivially satisfiable, and as we will see, this holds even if the clauses are only nearly pairwise disjoint, *i.e.*, if the intersection of the supports of each pair of clauses has size less than $2^k/e$.

This result has been well known as a consequence of the Lovász Local Lemma [14], whose original proof is non-constructive, and so does not produce a satisfying truth assignment (in polynomial time) when applied to an instance of k-SAT. Some efficient constructive solutions to k-SAT have been known, but only for suboptimal clause intersection sizes. Moser [26] first presented a constructive solution to k-SAT with near optimal clause intersection sizes, and Moser and Tardos [27] then generalized this result to the full Lovász Local Lemma for optimal clause intersection sizes. The analysis which we reproduce in this section comes from Fortnow's rephrasing of Moser's proof for k-SAT using the incompressibility method [16].

Moser's algorithm is remarkably naïve, and can be described in only a few sentences: Pick a uniformly random truth assignment for the variables of φ. For each unsatisfied clause, attempt to fix it by producing a new uniformly random truth assignment for its support, and recursively fix any intersecting clause which is made unsatisfied by this reassignment. We describe this process more carefully in the algorithms Solve and Fix below.

```
SOLVE(φ):
  α ← uniformly random truth assignment in {true, false}^n
  while φ(α) = false do
      D ← an unsatisfied clause in φ
      α ← FIX(φ, α, D)
  return α
```

```
FIX(φ, α, D):
  β ← uniformly random truth assignment in {true, false}^k
  replace the assignments in α for D's support with the values in β
  while φ(α) = false do
      D′ ← an unsatisfied clause in φ intersecting D
      α ← FIX(φ, α, D′)
  return α
```

Theorem 12. *Given a k-CNF formula φ with m clauses and n variables such that each clause intersects at most $r \leq 2^{k-3}$ other clauses, then the total number of invokations of* Fix *in the execution of* Solve(φ) *is at least $s + m \log m$ with probability at most 2^{-s}.*

Proof. Suppose that Fix is called $t = \lceil s + m \log m \rceil$ times. Let $\alpha \in \{\mathsf{true}, \mathsf{false}\}^n$ be the initial truth assignment for φ, and let $\beta_1, \ldots, \beta_t \in \{\mathsf{true}, \mathsf{false}\}^k$ be the local truth assignments produced in each call to Fix. The string $\gamma = (\alpha, \beta_1, \ldots, \beta_t)$ is uniformly chosen from a set of size 2^{n+tk}, and will be the subject of our encoding.

The execution of Solve(φ) determines a (rooted ordered) *recursion tree* T on $t+1$ nodes as follows: The root of T corresponds to the initial call to Solve(φ). Every other node corresponds to a call to Fix. The children of a node correspond to the sequence of calls to Fix that the procedure performs, ordered from left to right. Each (non-root) node in the tree is assigned a clause and its uniformly random truth assignment produced during the call to Fix. Moreover, a pre-order traversal of this tree describes the order of function calls in the algorithm's execution.

The string γ can be recovered in a bottom-up manner from our knowledge of the tree T and the final truth assignment α' after t calls to Fix. Specifically, let $D_1, \ldots, D_t$ be the clauses encountered in a pre-order traversal of T. In particular, D_t is the last fixed clause in the execution. Since D_t was not satisfied before its reassignment, this allows us to deduce k values of the previous assignment before D_t was fixed. Pruning D_t from the tree and continuing in this manner at D_{t-1}, we eventually recover the original truth assignment α produced in Solve(φ).

Therefore, to encode γ, we give the final truth assignment α'; and a description of the shape of the tree T; and the sequence of at most m clauses which are children of the root of T; and the at most t clauses involved in the calls to Fix in a pre-order traversal of T.

The key savings come from the fact that each clause intersects at most r other clauses, so each clause (which is not a child of the root) can be encoded using $\log r$ bits. Each clause which is a child of the root can be encoded using $\log m$ bits, and since the order of these children might be significant, we use $m \log m$ bits to encode the full sequence of these clauses. Finally, as in Lemma 5, the shape of T can be encoded using $2t$ bits. In total, the code has length

$$\begin{aligned}
|C(\gamma)| &\le n + 2t + m\log m + t\log r \\
&\le n + 2t + m\log m + t(k-3) && \text{(since } r \le 2^{k-3}\text{)} \\
&= n + tk - t + m\log m \\
&\le n + tk - s \enspace .
\end{aligned}$$

The result is obtained by applying the Uniform Encoding Lemma. □

Remark 9. By more carefully encoding of the shape of the recursion tree above, Messner and Thierauf [23] gave an encoding argument for the above result in which $r < 2^k/e$. Specifically, their refinement follows from a more careful counting of the number of trees with nodes of bounded degree.

4 The Non-Uniform Encoding Lemma and Shannon-Fano Codes

Thus far, we have focused on applications that could always be modelled as choosing some element x uniformly at random from a finite set X. To encompass even more applications, it is helpful to have an Encoding Lemma that deals with *non-uniform* distributions over X. First, we recall the following useful classic results:

Theorem 13 (Markov's Inequality)**.** *For any non-negative random variable Y with finite expectation, and any $a > 0$,*

$$\Pr\{Y \ge a\} \le (1/a)\,\mathrm{E}\{Y\} \enspace .$$

We will say that a real-valued function $\ell : X \to \mathbb{R}$ *satisfies Kraft's condition* if

$$\sum_{x\in X} 2^{-\ell(x)} \le 1 \enspace .$$

Lemma 7 (Kraft's Inequality)**.** *If $C : X \nrightarrow \{0,1\}^*$ is a partial prefix-free code, then the function $\ell : x \mapsto |C(x)|$ satisfies Kraft's condition. Conversely, for any function $\ell : X \to \mathbb{N}$ satisfying Kraft's condition, there exists a prefix-free code $C : X \to \{0,1\}^*$ such that $|C(x)| = \ell(x)$ for all $x \in X$.*

The following generalization of the Uniform Encoding Lemma, which was originally proven by Barron [3, Theorem 3.1], serves for non-uniform input distributions:

Lemma 8 (Non-Uniform Encoding Lemma)**.** *Let $C : X \nrightarrow \{0,1\}^*$ be a partial prefix-free code, and let p_x, $x \in X$, be a probability distribution on X. Suppose we draw $x \in X$ randomly according to p_x. Then, for any $s \ge 0$.*

$$\Pr\{|C(x)| \le \log(1/p_x) - s\} \le 2^{-s} \enspace .$$

Proof. We use Chernoff's trick, Markov's inequality, and Kraft's inequality, as follows:

$$\begin{aligned}
\Pr\{|C(x)| \le \log(1/p_x) - s\} &= \Pr\{|C(x)| - \log(1/p_x) \le -s\} \\
&= \Pr\{\log(1/p_x) - |C(x)| \ge s\} \\
&= \Pr\left\{2^{\log(1/p_x)-|C(x)|} \ge 2^s\right\} && \text{(Chernoff's trick)} \\
&\le \frac{\mathrm{E}\left\{2^{\log(1/p_x)-|C(x)|}\right\}}{2^s} && \text{(Markov's inequality)} \\
&= \frac{1}{2^s}\left(\sum_{x\in X:C(x)\ne\perp} p_x \cdot 2^{\log(1/p_x)-|C(x)|}\right) \\
&= \frac{1}{2^s}\left(\sum_{x\in X:C(x)\ne\perp} 2^{-|C(x)|}\right) \ .
\end{aligned}$$

By Kraft's inequality, $\sum_{x\in X:C(x)\ne\perp} 2^{-|C(x)|} \le 1$, and the result is obtained. □

The Non-Uniform Encoding Lemma is a strict generalization of the Uniform Encoding Lemma: Take $p_x = 1/|X|$ for all $x \in X$ and we obtain the Uniform Encoding Lemma.

As in Section 2.4, we will be interested in using a Shannon-Fano code C_α to encode Bernoulli(α) bit strings of length n. Recall that for such a string x, this code has length

$$n_1(x)\log(1/\alpha) + n_0(x)\log(1/(1-\alpha)) \ ,$$

since we are not concerned with ceilings.

5 Applications of the Non-Uniform Encoding Lemma

5.1 Chernoff Bound

We will now prove the so-called *additive version* of the Chernoff bound on the tail of a binomial random variable [8]. Theorem 2 established the special case of this result for Bernoulli(1/2) bit strings.

Theorem 14. *If B is a* Binomial(n,p) *random variable, then for any $\varepsilon \ge 0$,*

$$\Pr\{B \le (p-\varepsilon)n\} \le 2^{-nD(p-\varepsilon\,\|\,p)} \ ,$$

where

$$D(p\,\|\,q) = p\log(p/q) + (1-p)\log((1-p)/(1-q))$$

is the Kullback-Liebler divergence *or* relative entropy *between* Bernoulli(p) *and* Bernoulli(q) *random variables.*

Proof. By definition, $B = \sum_{i=1}^{n} x_i$, where $x_1,\ldots,x_n$ are independent Bernoulli(p) random variables. We will use an encoding argument on the bit string $x = (x_1,\ldots,x_n)$. The proof is almost identical to that of Theorem 2—now, we encode x using a Shannon-Fano code C_α, with $\alpha = p - \varepsilon$. Such a code has length

$$|C_{p-\varepsilon}(x)| = n_1(x)\log(1/(p-\varepsilon)) + n_0(x)\log(1/(1-p+\varepsilon)) \ .$$

Now, x appears with probability $p_x = p^{n_1(x)}(1-p)^{n_0(x)}$, so

$$\begin{aligned}
|C_{p-\varepsilon}(x)| &= \log(1/p_x) + n_1(x)\log(p/(p-\varepsilon)) + (n-n_1(x))\log((1-p)/(1-p+\varepsilon)) \\
&= \log(1/p_x) + n_1(x)\log\left(1+\frac{\varepsilon}{p-\varepsilon}\right) + (n_1(x)-n)\log\left(1+\frac{\varepsilon}{1-p}\right) \ ,
\end{aligned}$$

and $|C_{p-\varepsilon}(x)|$ increases as a function of $n_1(x)$. Therefore, if $n_1(x) \le (p-\varepsilon)n$, then

$$\begin{aligned}
|C_{p-\varepsilon}(x)| &\le \log(1/p_x) - n(p-\varepsilon)\log((p-\varepsilon)/p) - n(1-p+\varepsilon)\log((1-p+\varepsilon)/(1-p)) \\
&= \log(1/p_x) - nD(p-\varepsilon \,\|\, p) \ .
\end{aligned}$$

The Chernoff bound is obtained by applying the Non-Uniform Encoding Lemma. □

5.2 Percolation on the Torus

Percolation theory studies the emergence of large components in random graphs. For a general study of percolation theory, see the book by Grimmett [17]. We give an encoding argument proving that percolation occurs on the torus when edge survival rate is greater than 2/3, *i.e.* in random subgraphs of the torus grid graph in which each edge is included independently at random with probability at least 2/3, only at most one large component emerges. Our line of reasoning follows what is known as a *Peierls argument*.

Suppose that $\sqrt{n}$ is an integer. The $\sqrt{n} \times \sqrt{n}$ *torus grid graph* is defined to be the graph with vertex set $\{1,\ldots,\sqrt{n}\}^2$, where (i,j) is adjacent to (k,l) if

- $|i-k| \equiv 1 \pmod{\sqrt{n}}$ and $|j-l| = 0$, or
- $|i-k| = 0$ and $|j-l| \equiv 1 \pmod{\sqrt{n}}$.

Theorem 15. *Suppose that $\sqrt{n}$ is an integer. Let G be a subgraph of the $\sqrt{n}\times\sqrt{n}$ torus grid graph in which each edge is chosen with probability $p < 1/3$. Then, the probability that G contains a cycle of length at least*

$$\frac{s + \log n + O(1)}{\log(1/(3p))}$$

is at most 2^{-s}.

Proof. Let A be the bitstring of length $2n$ encoding the edge set of G. then the probability p_G that teh graph G is sampled is

$$p_G = p^{n_1(A)}(1-p)^{n_0(A)} \ .$$

Suppose that G contains a cycle C' of length $t \geq (s + \log n + O(1))/\log(1/(3p))$. Encode A by giving a single vertex u in C'; the sequence of directions that the cycle moves along from u; and a Shannon-Fano code with parameter p for the remaining edges of G.

There are four possibilities for the direction of the first step taken by C' from u, but only three for each subsequent choice. Thus, this sequence can be specified by $2+(t-1)\log 3$ bits. The total length of our code is then

$$\begin{aligned}|C(G)| &= \log n + 2 + (t-1)\log 3 + (n_1(A) - t)\log(1/p) + n_0(A)\log(1/(1-p)) \\ &= \log(1/p_G) + \log n - t\log(1/(3p)) + O(1) \\ &\leq \log(1/p_G) - s\end{aligned}$$

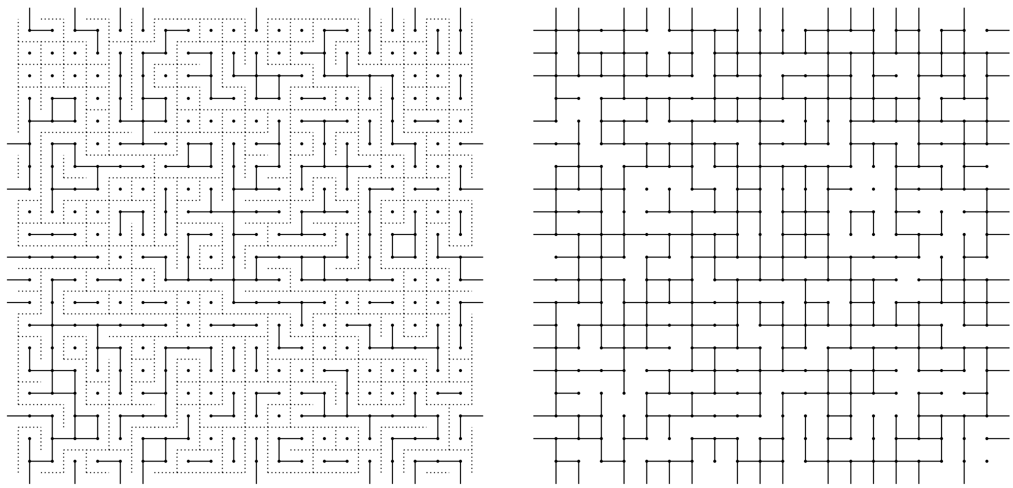
by our choice of t. We finish by applying the Non-Uniform Encoding Lemma. □

The torus grid graph can be drawn in the obvious way without crossings on the surface of a torus. This graph drawing gives rise to a dual graph, in which each vertex corresponds to a face in the primal drawing, and two vertices are adjacent if and only their primal faces are incident to the same edge. This dual graph is isomporphic to the original torus grid graph.

The obvious drawing of the torus grid graph also induces drawings for any of its subgraphs. Such a subgraph also has a dual, where each vertex corresponds to a face in the dual torus grid graph, and two vertices are adjacent if and only if their corresponding faces are incident to the same edge of the original subgraph.

Theorem 16. *Suppose that $\sqrt{n}$ is an integer. Let G be a subgraph of the $\sqrt{n} \times \sqrt{n}$ torus grid graph in which each edge is chosen with probability greater than $2/3$. Then, G has at most one component of size $\omega(\log^2 n)$ with high probability.*

Proof. See Figure 6 for a visualization of this phenomenon. Suppose that G has at least two components of size $\omega(\log^2 n)$. Then, there is a cycle of faces separating these components whose length is $\omega(\log n)$. From the discussion above, such a cycle corresponds to a cycle of $\omega(\log n)$ missing edges in the dual graph, as in Figure 6a. From Theorem 15, we know that this does not happen with high probability. □



(a) When $p = 0.33 < 1/3$, long cycles are rare. Dotted lines show missing edges in the dual.

(b) When $p = 0.67 > 2/3$, there is likely only one large component.

Figure 6: Random subgraphs of the 20×20 torus grid graph.

5.3 Triangles in $G_{n,p}$

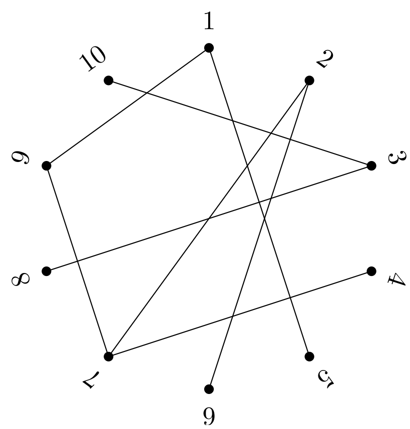
Recall as in Section 3.1 that the Erdős-Rényi random graph $G_{n,p}$ is the probability space of undirected graphs with vertex set $V = \{1,\ldots,n\}$ and in which each edge $\{u,w\} \in \binom{V}{2}$ is present with probability p and absent with probability $1-p$, independently of the other edges.

By linearity of expectation, the expected number of triangles (cycles of length 3) in $G_{n,p}$ is $p^3\binom{n}{3}$. For $p = (6c)^{1/3}/n$, this expectation is $c - O(1/n)$. Unfortunately, even when c is a large constant, it still takes some work to show that there is a constant probability that $G_{n,p}$ contains at least one triangle. Indeed, this typically requires the use of the second moment method, which involves computing the variance of the number of triangles in $G_{n,p}$. To show that $G_{n,p}$ has a triangle with more significant probability is even more complicated, and a proof of this result would still typically rely on an advanced probabilistic inequality [1]. Here we show how this can be accomplished with an encoding argument.

Theorem 17. *For $c > 0$ and $p = c/n$, $G \in G_{n,p}$ contains at least one triangle with probability at least $1 - 2^{-\Omega(c^3)}$.*

Proof. In this argument, we will produce an encoding of G's adjacency matrix, A. For simplicity of exposition, we assume that n is even.

Refer to Figure 7. If G contains no triangles, then we look at the number of ones in



	1	2	3	4	5	6	7	8	9	10
1		0	0	0	1	0	0	0	1	0
2			0	0	0	1	1	0	0	0
3				0	0	0	0	1	0	1
4					0	0	1	0	0	0
5						0	0	0	0	0
6							0	0	0	0
7								0	1	0
8									0	0
9										0

Figure 7: The random graph $G_{n,c/n}$ contains triangles when c is large enough. The highlighted 0 bits in the last five rows can be deduced from pairs of 1 bits in the first 5 rows.

the $n/2 \times n/2$ submatrix M determined by rows $1,\ldots,n/2$ and columns $n/2+1,\ldots,n$. Note that $n_1(M)$, the number of ones in M, is a Binomial$(n^2/4, c/n)$ random variable. There are two cases to consider:

0. The number of ones in M is at most $cn/8$. In this case, the number of ones in this submatrix is much less than the expected number, $cn/4$. Here one can apply the same argument used to prove Chernoff's bound (Theorem 14) or simply apply Chernoff's bound. We leave this as an exercise to the reader.

1. The number of ones in the submatrix is greater than $cn/8$. Notice that, for $i < j < k$ if $A_{i,j} = 1$ and $A_{i,k} = 1$, then the fact that there are no triangles implies that $A_{j,k} = 0$.

   Let m_i be the number of ones in the i-th row of the submatrix. By specifying rows $1,\ldots,n/2$, we eliminate the need to specify

$$m = \sum_{i=1}^{n/2} \binom{m_i}{2} \geq (n/2)\binom{2n_1(M)/n}{2} \geq (n/2)\binom{c/4}{2} = \Omega(c^2 n) \ ,$$

   zeros in rows $n/2+1,\ldots,n$ (here, we used the fact that the function $x \mapsto \binom{x}{2}$ is convex and increasing for $x \geq 1/4$). We thus encode G by giving a Shannon-Fano code with parameter p for the first $n/2$ rows of A; and a Shannon-Fano code with parameter p for the rest of A, excluding the bits which can be deduced from the preceding information. Such a code has length

$$|C(G)| = n_1(A)\log(1/p) + (n_0(A) - m)\log(1/(1-p))$$

   which results in a savings of

$$s = \log(1/p_G) - |C(G)| = m\log(1/(1-p)) \geq \Omega(c^2 n)\log(1/(1-p)) = \Omega(c^3) \ . \qquad \square$$

Theorem 18. *When $p = 1/(\alpha n)$ with $\alpha > 0$, then $G \in G_{n,p}$ has no triangle with probability at least $1 - \alpha^{-3}$.*

Proof. Suppose that G contains a triangle. We encode the adjacency matrix A of G. First, we specify the triangle's vertices; and finish with a Shannon-Fano code with parameter p for the remaining vertices of the graph. This code has length

$$\begin{aligned}
|C(G)| &= 3\log n + (n_1(A) - 3)\log(1/p) + n_0(A)\log(1/(1-p)) \\
&= \log(1/p_G) + 3\log n - 3\log(1/p) \\
&= \log(1/p_G) - 3\log\alpha \\
&= \log(1/p_G) - \log\alpha^3 \quad . \qquad \square
\end{aligned}$$

Together, Theorem 17 and Theorem 18 establish the fact that $1/n$ is a *threshold function* for triangle-freeness, *i.e.* if $p = o(1/n)$, then $G \in G_{n,p}$ has no triangle with high probability, and if $p = \omega(1/n)$, then G has a triangle with high probability.

6 Encoding with Kraft's Condition

As promised in Section 2.3, we finally discuss why it has made sense to omit ceilings in all of our encoding arguments.

Let $[0,\infty]$ denote the set of extended non-negative real numbers, supporting the extended arithmetic operations $a + \infty = \infty$ for all $a \in [0,\infty]$, and $2^{-\infty} = 0$.

Recall from Section 4 that a function $\ell : X \to [0,\infty]$ *satisfies Kraft's condition* if

$$\sum_{x \in X} 2^{-\ell(x)} \le 1 \quad .$$

The main observation is that neither the (Non-)Uniform Encoding Lemma nor any of its applications has actually required the specification of an explicit prefix-free code: We know, by construction, that every code we have presented is prefix-free, but we could also deduce from Kraft's inequality that, since our described codes satisfy Kraft's condition, a prefix-free code with the same codeword lengths exists. Similarly, we will see that it is actually enough to assign to every element from our universe a codeword *length* such that Kraft's condition is satisfied. These codeword lengths need not be integers.

Lemma 9. *Let $\ell : X \to [0,\infty]$ satisfy Kraft's condition and let $x \in X$ be drawn randomly where $p_x > 0$ denotes the probability of drawing x. Then*

$$\Pr\{\ell(x) \le \log(1/p_x) - s\} \le 2^{-s} \quad .$$

Proof. The proof is identical to that of Lemma 8. $\square$

The *sum* of two functions $\ell : X \to [0,\infty]$ and $\ell' : X' \to [0,\infty]$ is the function $\ell + \ell' : X \times X' \to [0,\infty]$ defined by $(\ell + \ell')(x,x') = \ell(x) + \ell'(x')$. Note that for any partial codes $C : X \nrightarrow \{0,1\}^*, C' : X' \nrightarrow \{0,1\}^*$, any $x \in X$, and any $x' \in X'$,

$$(|C| + |C'|)(x,x') = |C(x)| + |C'(x')| = |(C \cdot C')|(x,x') \ .$$

In other words, the sum of the functions of codeword lengths describes the length of codewords in concatenated codes.

Lemma 10. *If $\ell : X \to [0,\infty]$ and $\ell' : X' \to [0,\infty]$ satisfy Kraft's condition, then so does $\ell + \ell'$.*

Proof. Kraft's condition still holds:

$$\sum_{(x,x') \in X \times X'} 2^{-(\ell+\ell')(x,x')} = \sum_{x \in X} \sum_{x' \in X'} 2^{-\ell(x) - \ell'(x')} = \sum_{x \in X} 2^{-\ell(x)} \sum_{x' \in X'} 2^{-\ell'(x')} \leq 1 \ . \qquad \square$$

This is analogous to the fact that the concatenation of prefix-free codes is prefix-free.

Lemma 11. *For any probability density $p : X \to (0,1)$, the function $\ell : X \to [0,\infty]$ with $\ell(x) = \log(1/p_x)$ satisfies Kraft's condition.*

Proof.

$$\sum_{x \in X} 2^{-\ell(x)} = \sum_{x \in X} 2^{-\log(1/p_x)} = \sum_{x \in X} p_x = 1 \ . \qquad \square$$

This tells us that we can ignore the ceiling in every instance of a fixed-length code and every instance of a Shannon-Fano code while encoding.

We now give a tight notion corresponding to Elias codes.

Theorem 19 (Beigel [4])**.** *Fix some $0 < \varepsilon < e - 1$. Let $\ell : \mathbb{N} \to \mathbb{R}$ be defined as*

$$\ell(i) = \log i + \log\log i + \cdots + \underbrace{\log \cdots \log}_{\log^* i \text{ times}} i - (\log\log(e - \varepsilon))\log^* i + O(1) \ .$$

Then, ℓ satisfies Kraft's condition. Moreover, the function $\ell' : \mathbb{N} \to \mathbb{R}$ with

$$\ell'(i) = \log i + \log\log i + \cdots + \underbrace{\log \cdots \log}_{\log^* i \text{ times}} i - (\log\log e)\log^* i + c$$

does not satisfy Kraft's condition for any choice of the constant c.

It is not hard to see how Lemma 9, Lemma 10, Lemma 11, and Theorem 19 can be used to give encoding arguments with real-valued codeword lengths. For example, recall how the result of Theorem 1 carried an artifact of binary encoding. Using our new tools, we can now refine this and recover the exact result.

Theorem 1b. *Let $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ be chosen uniformly at random and let $t = \lceil \log n + s \rceil$. Then, the probability that there exists an $i \in \{1, \ldots, n-t+1\}$ such that $x_i = x_{i+1} = \cdots = x_{i+t-1} = 1$ is at most 2^{-s}.*

Proof. Let $\ell : \{0,1\}^n \to [0,\infty]$ be such that if x contains a run of $t = \lceil \log n + s \rceil$ ones, then $\ell(x) = \log n + n - t$, and otherwise $\ell(x) = \infty$. We will show that ℓ satisfies Kraft's condition.

Let the function $f : \{1, \ldots, n-t+1\} \to [0,\infty]$ have $f(i) = \log n$ for all $i \in \{1, \ldots, n-t+1\}$, and $g : \{0,1\}^{n-t} \to [0,\infty]$ have $g(y) = n - t$ for all $y \in \{0,1\}^{n-t}$. Both f and g satisfy Kraft's condition by Lemma 11. By Lemma 10, so does the function

$$h = f + g : \{1, \ldots, n-t+1\} \times \{0,1\}^{n-t} \to [0,\infty] \ ,$$

where $h(i,y) = \log n + n - t$ for all i and y. Crucially, each element of the set $\{1, \ldots, n-t+1\} \times \{0,1\}^{n-t}$ corresponds to an n-bit binary string containing a run of t ones: the element $(i,y) \in \{1, \ldots, n-t+1\} \times \{0,1\}^{n-t}$, where $y = (y_1, \ldots, y_{n-t})$, corresponds to the binary string

$$(y_1, \ldots, y_{i-1}, \underbrace{1, 1, \ldots, 1}_{t \text{ times}}, y_i, \ldots, y_{n-t}) \ .$$

Therefore, ℓ satisfies Kraft's condition. By our choice of t, we have that $\ell(x) \le n - s$ if and only if x contains a run of t ones. We finish by applying Lemma 9. □

7 Summary and Conclusions

We have described a simple method for producing encoding arguments. Using our encoding lemmas, we gave original proofs for several previously established results.

Typically, one would invoke the incompressibility method after developing some of the theory of Kolmogorov complexity. Our technique requires only a basic understanding of prefix-free codes and one simple lemma. We are also the first to suggest a simple and tight manner of encoding using only Kraft's condition with real-valued codeword lengths. In this light, we posit that there is no reason to develop an encoding argument through the incompressibility method: our Uniform Encoding Lemma is simpler, the Non-Uniform Encoding Lemma is more general, and our technique from Section 6 is less wasteful. Indeed, though it would be easy to state and prove our Non-Uniform Encoding Lemma in the setting of Kolmogorov complexity, it seems as if the general encoding lemma from Section 6 only can exist in our simplified framework.

Acknowledgements

This research was initiated in response to an invitation for the first author to give a talk at the Ninth Annual Probability, Combinatorics and Geometry Workshop, held April 4–11 at McGill University's Bellairs Institute. Many ideas that appear in the current paper were developed during the workshop. The author is grateful to the other workshop participants for providing a stimulating working environment. In particular, Xing Shi Cai pointed out the application to runs in binary strings (Theorem 1) and Gábor Lugosi stated and proved the Non-Uniform Encoding Lemma (Lemma 8) in response to the first author's half-formed ideas about a non-uniform version of Lemma 1, and then later pointed us to the proof in Barron's thesis. We would also like to thank Günter Rote for valuable comments.

References

[1] N. Alon and J. H. Spencer. *The Probabilistic Method*. Wiley Series in Discrete Mathematics and Optimization. Wiley-Interscience, 2 edition, 2004.

[2] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, February 2000.

[3] A. R. Barron. *Logically Smooth Density Estimation*. PhD thesis, Stanford University, September 1985.

[4] R. Beigel. Unbounded searching algorithms. *SIAM Journal on Computing*, 19(3):522–537, 1990.

[5] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. *SIAM Journal on Computing*, 35(6):1350–1385, 2006.

[6] R. Boucheron, G. Lugosi, and P. Massart. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford University Press, Oxford, United Kingdom, 2013.

[7] A. Cayley. A theorem on trees. *Quarterly Journal of Mathematics*, 23:376–378, 1889.

[8] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 12 1952.

[9] L. Devroye. Applications of the theory of records in the study of random trees. *Acta Informatica*, 26(1):123–130, 1988.

[10] L. Devroye, P. Morin, and A. Viola. On worst-case Robin Hood hashing. *SIAM Journal on Computing*, 33(4):923–936, 2004.

[11] D. P. Dubhashi and A. Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, New York, New York, 2009.

[12] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975.

[13] P. Erdős. Some remarks on the theory of graphs. *Bulletin of the American Mathematics Society*, 53:292–294, 1947.

[14] P. Erdős and L. Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In A. Hajnal, R. Rado, and V. T. Sós, editors, *Infinite and Finite Sets*, volume 10 of *Colloquia Mathematica Societatis János Bolyai*, pages 609–627. North-Holland, 1973.

[15] R. M. Fano. The transmission of information. Technical Report 65, Research Laboratory of Electronics at MIT, Cambridge, Massachusetts, 1949.

[16] L. Fortnow. A Kolmogorov complexity proof of the Lovász local lemma. http://blog.computationalcomplexity.org/2009/06/kolmogorov-complexity-proof-of-lov.html. (last accessed November 28, 2015).

[17] G. R. Grimmett. *Percolation*. 321. Springer-Verlag Berlin Heidelberg, 2 edition, 1999.

[18] Terese Haimberger. Theoretische und Experimentelle Untersuchung von Kuckucks-Hashing. Bachelor's thesis. Freie Universität Berlin, 2013.

[19] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, July 1961.

[20] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, October 2006.

[21] B. Lucier, T. Jiang, and M. Li. Average-case analysis of Quicksort and binary insertion tree height using incompressibility. *Information Processing Letters*, 103(2):45–51, July 2007.

[22] C. McDiarmid. On the method of bounded differences. In J. Siemons, editor, *Surveys in Combinatorics: Invited Papers at the 12th British Combinatorial Conference*, number 141, pages 148–188. Cambridge University Press, 1989.

[23] J. Messner and T. Thierauf. A Kolmogorov complexity proof of the Lovász local lemma for satisfiability. *Theoretical Computer Science*, 461:55–64, November 2012.

[24] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[25] P. Morin. *Open Data Structures: An Introduction*. Athabasca University Press, Edmonton, 2013. Also freely available at opendatastructures.org.

[26] R. A. Moser. A constructive proof of the Lovász local lemma. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, STOC '09, pages 343–350, New York, NY, USA, 2009. ACM.

[27] R. A. Moser and G. Tardos. A constructive proof of the general Lovász local lemma. *Journal of the ACM*, 57(2):11:1–11:15, January 2010.

[28] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, New York, 1995.

[29] ACM/IEEE-CS Joint Task Force on Computing Curricula. Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science. Technical report, ACM Press and IEEE Computer Society Press, December 2013.

[30] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[31] M. S. Pinsker. On the complexity of a concentrator. In *The 7th International Teletraffic Conference*, volume 4, pages 1–4, 1973.

[32] M. Pătraşcu. Cuckoo hashing. http://infoweekly.blogspot.ca/2010/02/cuckoo-hashing.html. (last accessed April 15, 2015).

[33] B. Reed. The height of a random binary search tree. *Journal of the ACM*, 50(3):306–332, May 2003.

[34] H. Robbins. A remark on Stirling's formula. *The American Mathematical Monthly*, 62(1):26–29, jan 1955.

[35] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[36] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423, 1948.

[37] P. Vitányi. Analysis of sorting algorithms by Kolmogorov complexity (a survey). In *Entropy, Search, Complexity*, volume 16 of *Bolyai Society Mathematical Studies*, pages 209–232. Springer-Verlag New York, 2007.

[38] B. Vöcking. How asymmetry helps load balancing. *Journal of the ACM*, 50(4):568–589, July 2003.