On the Uniformly Random Generation of Combinatorial Structures

By
Jordan Arthur Dempsey

A Thesis Presented to
Saint Mary's University, Halifax, Nova Scotia
in Partial Fulfillment of the Requirements for
the Degree of Bachelor of Science, Honours Computing Science.

April 29, 2019, Halifax, Nova Scotia

Copyright © Jordan Arthur Dempsey

Approved: Dr. John C. Irving

Associate Professor, Supervisor

Approved: Dr. Bert L. Hartnell

Professor, Reader

Date: April 29, 2019

On the Uniformly Random Generation of Combinatorial Structures

Jordan Arthur Dempsey

April 29, 2019

Abstract

We consider two algorithmic processes used to sample uniformly from a wide array of combinatorial classes. First, the recursive method which uses the recursive decomposition of a class in addition to tables of large integers which count the objects of a given size to generate objects uniformly at random. Second, the more recent Boltzmann model for sampling is considered. Boltzmann models rely on the use of the closed form generating function of a class and address the space constraints imposed by the recursive method with a time efficiency trade off. Implementations for three classes of objects (binary trees, integer partitions, and set partitions) are demonstrated for each method. Comparison and analysis of the two methods are discussed in addition to applications.

Acknowledgements

I would like to thank first and foremost my supervisor Dr. John Irving, without whom this thesis would not exist. John's guidance (and sense of humour) made getting through this challenge possible. I further thank John for inspiring my interest in combinatorics and for teaching me countless valuable skills for writing mathematical documents.

I would like to thank my reader Dr. Bert Hartnell for his valuable comments and suggestions for improving the quality of this thesis. Furthermore, I would like to thank Bert for inspiring my initial fascination with discrete mathematics and its inherent connections with computing science.

I would like to thank Dr. Paul Muir for all of his guidance over the years with respect to navigating my degree requirements. I would also like to thank Paul for providing insight into world of numerical and scientific computing.

I would also like to thank all of the faculty members in the department of mathematics and computing science for all of the knowledge they have provided over the years. As well I would like to thank the department secretary, Rose Daurie, for all she does to keep everything running smoothly. I would also like to acknowledge the science advising staff at Saint Mary's for all the assistance they provide.

I would also like to thank my bosses throughout my time at Saint Mary's, Vicki, Mike, and Janice. The flexibility and support you have all provided over the years has been greatly appreciated.

I would like to thank my parents, family, and close friends who have offered so much support over the course of my undergraduate career — I cannot thank-you enough.

Finally, to my fellow thesis students, Connor, Matt, and Will — it's been awesome! We finally made it!

Contents

List of Theorems, Definitions, and Examples

List of Algorithms

List of Figures

1 Introduction

The main objective of this research is to investigate algorithmic methods for the uniformly random generation of combinatorial objects. In particular two major algorithmic processes will be explored and implemented: the recursive method of Nijenhuis and Wilf [9] and the Boltzmann sampling method of Flajolet et al. [4]. Our goal is to compare and contrast various aspects of these two methods — primarily with respect to efficiency and applicability.

We shall begin our study by recalling some fundamental concepts and notation from enumerative combinatorics.

1.1 Combinatorial Classes and Generating Functions

Definition 1.1 (Combinatorial Classes)**.** A *combinatorial class* is a pair $\mathscr{C} = (S, |\cdot|)$ consisting of a set S together with a function $|\cdot| : S \longrightarrow \mathbb{N}$ such that $\mathscr{C}_n := \{\gamma \in S : |\gamma| = n\}$ is finite for every $n \in \mathbb{N}$. Said otherwise, a combinatorial class is a set of objects which have some prescribed notion of size. The added condition $|\mathscr{C}_n| < \infty$ is used to express the natural restriction that the number of objects of a given size must be finite.

We write $c_n := |\mathscr{C}_n|$ to denote the number of objects of size n contained in a class $\mathscr{C}$ and refer to $(c_0, c_1, ...)$ as the *counting sequence* of $\mathscr{C}$. Our classes will all contain a unique *empty object* that has size 0 so that $c_0 = 1$. We declare the existence of empty objects as a matter of convenience. We further note that, for the classes considered, every object of size n shall have a decomposition into n subobjects of size 1 which will be refered to as *atoms*. For certain classes $\mathscr{C}$, the atoms comprising any object $\gamma \in \mathscr{C}_n$ will further carry distinct labels $1, 2, ..., n$. In this case we refer to $\mathscr{C}$ as a *labelled* class, and otherwise we say $\mathscr{C}$ is an *unlabelled* class.

Example 1.1 (Unlabelled Class)**.** The class $\mathscr{S}$ of all finite length binary strings (sequences of 0s and 1s) is an unlabelled class. The size of such a string is its length and clearly we have $c_n = 2^n$. For this class the empty object is the empty string ϵ which has length 0 and is mandated to be unique. Each bit of a string is an atom, and since we do not distinguish one 0 from another nor one 1 from another, the class is unlabelled.

Example 1.2 (Labelled Class)**.** The class $\mathscr{P}$ of all permutations (or orderings) of canonical finite sets is a labelled class. The size of a permutation is simply the cardinality of the set and the empty object is ϵ (corresponding to permutations on $\emptyset$). Each element from the set is an atom and the elements are distinguishable.

The Ordinary Generating Function (OGF) for a class $\mathscr{C}$ is defined by

$$C(x) = \sum_{n=0}^{\infty} c_n x^n$$

and its **Exponential Generating Function (EGF)** is given by

$$\widehat{C}(x) = \sum_{n=0}^{\infty} c_n \frac{x^n}{n!}$$

Although the OGF and EGF are defined for any class, we use the OGF for the unlabelled classes and the EGF for the labelled case.

Example 1.3 (Generating Functions)**.** For the class $\mathscr{S}$ of all finite length binary strings of length n considered in Example 1.1, we get that the OGF is:

$$S(x) = 1 + 2x + 4x^2 + 8x^3 + ... = \sum_{n=0}^{\infty} 2^n x^n = \frac{1}{1-2x}.$$

For the class $\mathscr{P}$ of all permutations of finite sets shown in Example 1.2, we get that

the EGF is:

$$P(x) = \sum_{n=0}^{\infty} \frac{n!x^n}{n!} = \sum_{n=0}^{\infty} x^n = \frac{1}{1-x}.$$

Definition 1.2 (Disjoint Union of Classes)**.** Let $\mathscr{A}$, $\mathscr{B}$ be classes whose underlying sets are disjoint. We write $\mathscr{C} = \mathscr{A} + \mathscr{B}$ to signify that $\mathscr{C}$ is the *disjoint union* of $\mathscr{A}$ and $\mathscr{B}$. Let $|\cdot|_{\mathscr{A}}$, $|\cdot|_{\mathscr{B}}$ be the size functions for $\mathscr{A}$ and $\mathscr{B}$ respectively. Then the size of $\gamma \in \mathscr{C}$ is

$$|\gamma|_{\mathscr{C}} = \begin{cases} |\gamma|_{\mathscr{A}}, \text{ if } \gamma \in \mathscr{A} \\ |\gamma|_{\mathscr{B}}, \text{ if } \gamma \in \mathscr{B} \end{cases}$$

Lemma 1.1 (Sum Lemma)**.** *Let $\mathscr{C} = \mathscr{A} + \mathscr{B}$. Then the OGF of $\mathscr{C}$ is $C(x) = A(x) + B(x)$.*

Proof. By definition

$$C(x) = \sum_{\gamma \in \mathscr{C}} x^{|\gamma|} = \sum_{\gamma \in \mathscr{A}} x^{|\gamma|} + \sum_{\gamma \in \mathscr{B}} x^{|\gamma|}$$

and hence

$$C(x) = A(x) + B(x).$$

□

Corollary 1.2. *If $\mathscr{C} = \mathscr{A} + \mathscr{B}$ then $c_n = a_n + b_n$.*

Definition 1.3 (Cartesian Product of Classes)**.** We write $\mathscr{C} = \mathscr{A} \times \mathscr{B}$ to mean that $\mathscr{C}$ is the *Cartesian product* of the classes $\mathscr{A}$ and $\mathscr{B}$. In other words, we have that every $\gamma \in \mathscr{C}$ can be written as an ordered pair (α, β), where $\alpha \in \mathscr{A}$ and $\beta \in \mathscr{B}$. Let $|\cdot|_{\mathscr{A}}$, $|\cdot|_{\mathscr{B}}$ be the size functions for $\mathscr{A}$ and $\mathscr{B}$ respectively. Then the size of $\gamma \in \mathscr{C}$ is $|\gamma|_{\mathscr{C}} = |(\alpha, \beta)|_{\mathscr{C}} = |\alpha|_{\mathscr{A}} + |\beta|_{\mathscr{B}}$.

Lemma 1.3 (Product Lemma)**.** *Let $\mathscr{C} = \mathscr{A} \times \mathscr{B}$. Then the OGF of $\mathscr{C}$ is $C(x) = A(x) \cdot B(x)$.*

Proof. By definition

$$C(x) = \sum_{\gamma \in \mathscr{C}} x^{|\gamma|} = \sum_{(\alpha,\beta) \in \mathscr{A} \times \mathscr{B}} x^{|\alpha+\beta|}$$

and hence

$$C(x) = \sum_{\alpha \in \mathscr{A}} \sum_{\beta \in \mathscr{B}} x^{|\alpha+\beta|} = \sum_{\alpha \in \mathscr{A}} \sum_{\beta \in \mathscr{B}} x^{|\alpha|} x^{|\beta|} = \sum_{\alpha \in \mathscr{A}} x^{|\alpha|} \cdot \sum_{\beta \in \mathscr{B}} x^{|\beta|} = A(x) \cdot B(x)$$

□

Corollary 1.4. *If $\mathscr{C} = \mathscr{A} \times \mathscr{B}$ then $c_n = \sum_{i=0}^{n} a_{n-i} b_i$.*

1.2 Uniform Sampling

In order to demonstrate the difference between naive generation and uniform generation we will consider generating random binary strings of length n having no occurrence of the substring 11.

Of course a naive manner in which we could do this would be to select each bit as follows: flip a coin, if it lands on heads we write a 0, if it lands on tails we write a 1. We then continue with the added constraint that we are forced to write a 0 immediately following any 1. Clearly this method will only generate legal strings and it is capable of generating all such strings. There are 5 legal strings of length 3, 2 beginning with 1 and 3 beginning with 0. The strings 101 and 100 both occur with probability $\frac{1}{2} \cdot 1 \cdot \frac{1}{2} = \frac{1}{4}$. On the other hand, 001 and 000 occur with probability $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$ while 010 arises with probability $\frac{1}{2} \cdot \frac{1}{2} \cdot 1 = \frac{1}{4}$. In a uniform sampling method we would see each string being generated with equal probability $\frac{1}{5}$. Hence the issue is that strings are not generated with equal probability — in particular, strings beginning with 1 are overweighted with this method.

One could instead simply generate random binary strings without restrictions and discard all illegal strings. While this would result in sampling from a uniform

distribution, there remains the problem that it is infeasible for large n. A legal string can start with a 0 and then be followed by a legal string of length $n-1$. Otherwise it starts a 10 and is followed by a legal string of length $n-2$. Let c_n be the number of legal strings of length n. Since there is 1 legal string of length 0 (ϵ) and there are 2 legal strings of length 1 (0 and 1), the total number of legal strings of length n is given by the recurrence relation is $c_n = c_{n-1} + c_{n-2}$, (for $n \geq 2$) with initial conditions $c_0 = 1$ and $c_1 = 2$. This is a Fibonacci recurrence and note $c_n = f_{n+1}$. There are 2^n possible bitstrings of length n and it is well known that $c_n \sim \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$. Since $\frac{c_n}{2^n} \to 0$ as $n \to \infty$, the chance of obtaining a legal string with this method approaches 0 for large n.

Given some non-negative integer n, our aim is to draw a random object from $\mathscr{C}_n$ such that the probability of any $\gamma \in \mathscr{C}_n$ being generated is exactly $\frac{1}{c_n}$. Using the recursive method we obtain a uniform distribution on $\mathscr{C}_n$. With Boltzmann sampling we instead sample from a nonuniform distribution on the entirety of $\mathscr{C}$, wherein any $\gamma \in \mathscr{C}$ is generated with probability $\frac{|\gamma|}{C(x)}$ for a fixed value of x. If we condition the Boltzmann sampler to return an object only if it is of size n, then we obtain a uniform distribution on $\mathscr{C}_n$.

2 The Recursive Method

The recursive method, which first emerged in the 1970's, has been extensively studied over the years since its initial conception [9]. As is suggested by the nomenclature, the recursive method applies to combinatorial classes that possess recursive decompositions thus allowing c_n to be expressed in terms of prior values $c_0, ..., c_{n-1}$. The values $c_0, c_1, ..., c_{n-1}$ are pre-computed and used to assign probabilities to the branches of a recursive algorithm so as to ultimately have the method return an uniformly random object of size n.

Example 2.1 (Recursive Classes)**.** Reconsider the example of binary strings with no occurrence of substring 11 and the corresponding recurrence relation $c_n = c_{n-1} + c_{n-2}$. We get $1 = p_n + q_n$, where $p_n = \frac{c_{n-1}}{c_n}$ and $q_n = \frac{c_{n-2}}{c_n}$. We then generate strings of length n with weighted coin flips such that heads turns up with probability p_n and tails with probability q_n. If the coin lands on heads, we write down a 0 and recurse to generate a string of length $n-1$, and if it lands on tails, we write down 10 and recurse to generate string of length $n-2$. With this method all strings of length n arise with equal probability, $\frac{1}{c_n}$. For instance, the string $0100010 = 0(10)00(10)$ arises with probability $p_7 q_6 p_4 p_3 q_2 = \frac{c_6}{c_7} \cdot \frac{c_4}{c_6} \cdot \frac{c_3}{c_4} \cdot \frac{c_2}{c_3} \cdot \frac{c_0}{c_2} = \frac{c_0}{c_7} = \frac{1}{c_7}$.

The main computationally intensive aspect of the recursive method lies within the need to pre-compute the large array of values $c_0, c_1, ..., c_n$ in order to generate random objects of size n. If the sequence $\{c_k\}$ is exponential (or worse) then the array of integers can easily overcome space constraints for large n, even on modern machines. For example if we wanted to generate uniformly random binary trees on n internal nodes (see Section 2.1) we must compute the *Catalan numbers* up to and including the n^{th} term. The n^{th} Catalan number is $c_n = \frac{1}{n+1}\binom{2n}{n}$. It was demonstrated in [12] that

$$c_n \sim \frac{4^n}{n^{\frac{3}{2}}\sqrt{\pi}}$$

hence the Catalan numbers grow exponentially fast. Indeed the 1000000^{th} Catalan number contains 602051 digits, and thus we can see that the tables of values required for the recursive method can exceed the space limitations of even modern computers.

2.1 Binary Trees

Definition 2.1 (Binary Trees)**.** A *tree* is an acyclic, connected graph. The acylic and connected properties of a tree T imply that there exists a unique path between any two vertices $u, v \in T$. We say that a tree is *rooted* if one of its vertices is distinguished

from the others as the *root*. Thus there is a unique path that exists between the root and each vertex of a rooted tree. We call u a *child* of v if v is the immediate neighbour of u on the path from u to the root. The number of children a vertex has is referred to as its *out-degree*. We refer to a vertex with out-degree 0 as a *leaf* and a vertex with out-degree ≥ 1 as an *internal node*. An *ordered tree* is a rooted tree that has an ordering imposed on the children of each vertex. Typically we do this by embedding a tree in the plane by placing the root at the top of the tree and then ordering the children of a vertex from left to right. A *binary tree* is an ordered tree where every vertex has either out-degree 0 or 2.

For example, the 5 binary trees on 3 internal nodes are depicted in Figure 2.1.

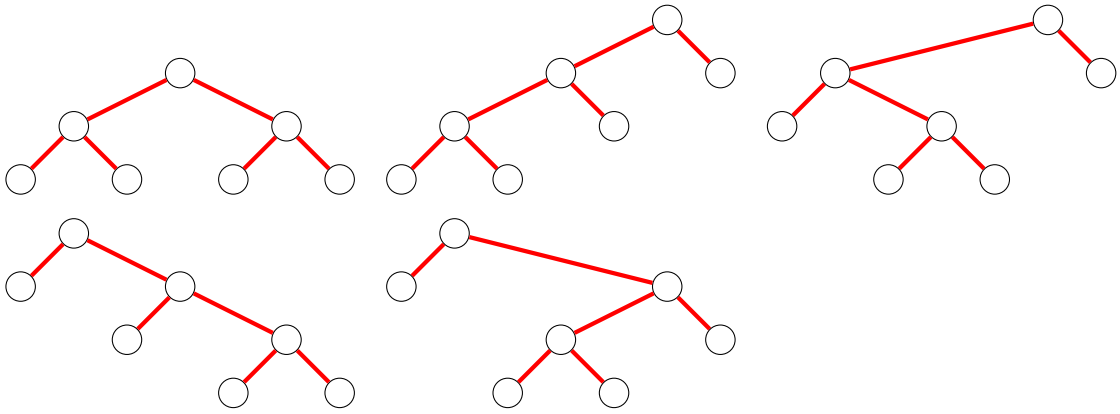

Figure 2.1: The 5 binary trees on 3 internal nodes.

Let $\mathscr{B}$ be the set of binary trees. Since we can easily see that a binary tree $T \in \mathscr{B}$ with n internal nodes has $n+1$ leaves, it makes sense to measure the size of a binary tree by its number of internal nodes. In other words, we view internal nodes as the atomic objects for the class $\mathscr{B}$, and the empty object ϵ is the tree comprised of a single leaf (the root vertex having no children).

If we remove the root vertex from a tree $T \in \mathscr{B}$, we are left with a pair $(T_L, T_R) \in \mathscr{B}^2$, consisting of the left and right principal subtrees of T. Thus we have the set

decomposition $\mathscr{B} = \{\circ\} \cup (\{\bullet\} \times \mathscr{B}^2)$, where $\circ$ denotes a leaf and $\bullet$ denotes an internal node.

Let $B = B(x) = \sum_{n=0}^{\infty} b_n x^n$ be the OGF of $\mathscr{B}$. By Lemma 1.1 and Lemma 1.3 we obtain $B = 1 + xB^2$. By virtue of the quadratic formula, we obtain:

$$xB^2 - B - 1 = 0$$
$$B = \frac{1 \pm \sqrt{1 - 4x}}{2x}.$$

We discard the positive branch as it is not defined at $x = 0$ and therefore has no power series expansion, leaving

$$B(x) = \sum_{n=0}^{\infty} b_n x^n = \frac{1 - \sqrt{1 - 4x}}{2x}.$$

Using the binomial theorem to expand $(1 - 4x)^{\frac{1}{2}}$ gives

$$B(x) = \sum_{n=0}^{\infty} \frac{1}{n+1} \binom{2n}{n} x^n,$$

and hence $b_n = \frac{1}{n+1}\binom{2n}{n}$. This demonstrates the well known fact that binary trees on n internal nodes are counted by the *Catalan numbers*.

Using our previous result that $B = 1 + xB^2$, we apply coefficient extraction on x^n to obtain the Catalan recursion

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-k-1}$$

with initial condition $b_0 = 1$. Dividing both sides by b_n we get:

$$1 = \sum_{k=0}^{n-1} \frac{b_k b_{n-k-1}}{b_n}$$

We recognize that the summand $\frac{b_k b_{n-k-1}}{b_n}$ gives the probability that a tree $T \in \mathscr{B}$ has principal subtrees of size k and size $n-k-1$. Hence we have the decomposition and probabilistic requirements necessary to develop a sampler with the recursive method [9].

Example 2.2 (Recursive Method Sampler for Binary Trees)**.** A pseudo-code implementation is displayed in Algorithm 2.1. The algorithm takes as input n, the desired number of internal nodes, and returns a binary tree $T \in \mathscr{B}_n$ which is selected uniformly at random from binary trees of size n. We observe that lines 8 through 14 implement the probability distribution, first we choose a random integer between 1 and the n^{th} Catalan number and initialize tally to 0. For each integer $i \in [0, n]$ we increase the tally by $b_i b_{n-i-1}$, once the value of x is less than the tally we set $k = i$, break from the loop, and finally in line 15 we return a binary tree with left principal subtree of size k and right principal subtree of size $n-k-1$, BinaryTree(0, 0) returns internal node whose children are leaves. When $n = 0$, a terminal leaf node is reached, thus allowing the function to terminate. The implementation of Algorithm 2.1 is shown in Appendix A.1.

Algorithm 2.1: Recursive method binary tree sampler.

Data: N - the desired number of internal nodes
Result: a randomly generated binary tree on n internal nodes

```
1  Compute Catalan numbers up to any including the n^th term and store in
     array b.
2  function genBinaryTree(n):
3  if n = 0 then
4  |   return ∘
5  else
6  |   x ← random integer between 1 and b_n
7  |   tally ← 0
8  |   for i from 0 to n do
9  |   |   tally ← tally + b_i b_{n−i−1}
10 |   |   if x ≤ tally then
11 |   |   |   k ← i
12 |   |   |   break
13 |   return BinaryTree(genBinaryTree(k), genBinaryTree(n − k − 1))
```

Illustrated in Figures 2.2 and 2.3 below are sample outputs of randomly generated binary trees on 100 and 200 internal nodes respectively, using the recursive method sampler.

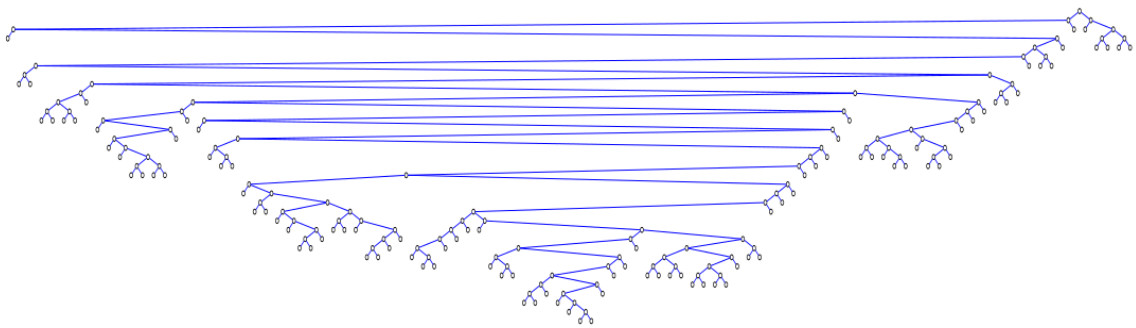

Figure 2.2: Random binary tree on 100 internal nodes.

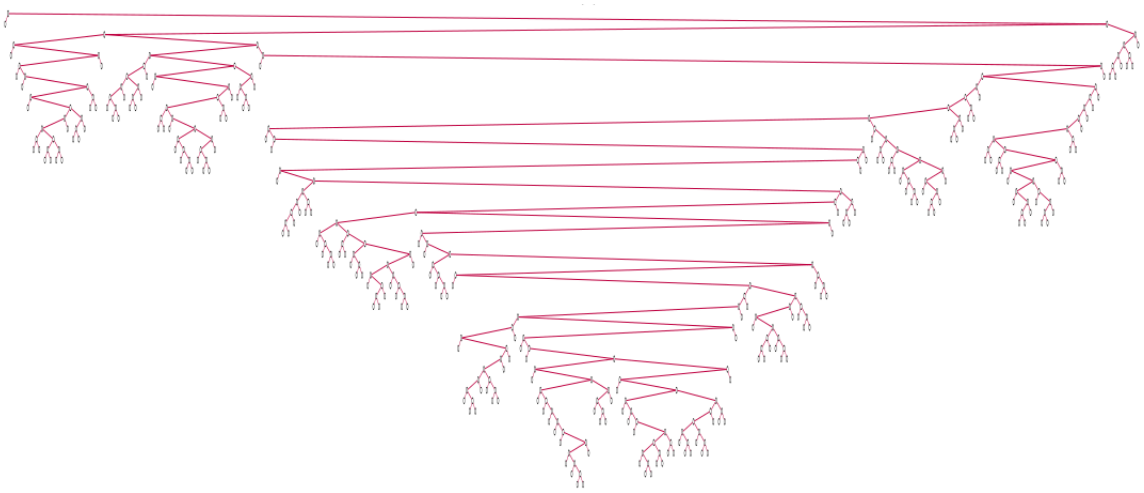

Figure 2.3: Random binary tree on 200 internal nodes.

We also note that there exist bijections from binary trees on n internal nodes and various other objects counted by the Catalan numbers. In principle this implies that a sampler for binary trees can be used to generate other classes of objects at random, however in practice this may not be practical.

2.2 Integer Partitions

Definition 2.2 (Integer Partitions)**.** A *partition* of $n \in \mathbb{N}$ is a weakly decreasing list $\lambda = (\lambda_1, \lambda_2, ..., \lambda_k)$ of positive integers such that $\sum_{i=1}^{k} \lambda_i = n$. The entries λ_i are referred to as the *parts* of λ.

We illustrate partitions with rows of boxes called *Young diagrams*, where the number of boxes in row i corresponds to the value of the i^{th} element of the partition. For instance, the 7 partitions of 5 are: (5), (4, 1), (3, 2), (3, 1, 1), (2, 2, 1), (2, 1, 1, 1), and (1, 1, 1, 1, 1), which are represented in Figure 2.4 with their corresponding Young diagrams.

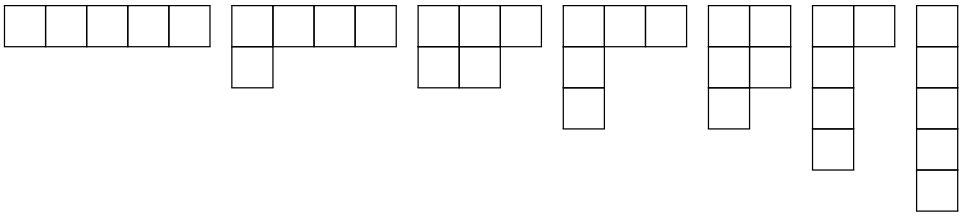

Figure 2.4: The seven partitions of $n = 5$.

We measure the size of a partition by the sum of its parts. The boxes of the Young diagram are the atomic objects and we allow for the unique partition of 0, denoted ϵ, which is deemed to have 0 parts. Note in Figure 2.4, each Young diagram is of size 5 and is composed of 5 atoms. Let $\mathscr{P}$ be the class of partitions and consider subclasses $\mathscr{P}_k$ of $\mathscr{P}$ consisting of partitions whose parts are at most k. We obtain the decomposition $\mathscr{P}_k = (\{k\} \times \mathscr{P}_k) \cup \mathscr{P}_{k-1}$ for $k \geq 1$, with $\mathscr{P}_0 = \{\epsilon\}$. This decomposition holds because any partition $\lambda \in \mathscr{P}_k$ either has a part equal to k or does not. If such a part exists then we remove it to get a partition of $n-k$ with parts at most k. Otherwise all parts of λ are at most $k-1$ and so $\lambda \in \mathscr{P}_{k-1}$. Let $p(n,k)$ be the number of partitions of n with parts at most k. Then by the decomposition we have

$$p(n,k) = p(n-k,k) + p(n,k-1)$$

for $n \geq k \geq 1$, with boundary conditions $p(0,k) = 1$ for $k \geq 0$ and $p(n,0) = 0$ for $n \geq 1$. Interpreting this recursion, we can develop a recursive method sampler for integer partitions.

Example 2.3 (Recursive Method Sampler for Integer Partitions)**.** A pseudo-code implementation is displayed in Algorithm 2.2. The algorithm takes as inputs n, the integer to partition, k, the maximum part size permitted, and λ, the list of parts. To sample uniformly from all partitions of n, we initially set $k = n$ and $\lambda = \{\epsilon\}$. In line 5 we select a random value between 0 and 1 and then in line 6 we let x be the probability that a partition of n with parts at most k has a part equal to k. Then in

lines 7 through 11 we have a probability distribution where we assign a part of size k to the partition with probability x and recurse on partitions of $n-k$ with parts at most k. Otherwise, with probability $1-x$, we recurse allowing parts of at most size $k-1$. Once the first parameter to the function equals 0, the function terminates and the partition λ of n is returned (lines 3 and 4). The implementation of Algorithm 2.2 is shown in Appendix A.2.

Algorithm 2.2: Recursive method integer partition sampler.

```
   Data: N - the integer to partition
   Result: a randomly generated partition of n
 1 Compute table of p(n,k) values for 0 ≤ k ≤ n.
 2 function genPart(n, k, λ):
 3 if n = 0 then
 4 |   return λ
 5 u ← random ∈ (0,1)
 6 x ← p(n−k,k) / p(n,k)
 7 if u < x then
 8 |   append k to λ
 9 |   return genPart(n − k, k, λ)
10 else
11 |   return genPart(n, k − 1, λ)
```

Illustrated below in Figure 2.5 we see sample output from the recursive method sampler for integer partitions of $n = 100$, 500, 1000, and 5000 respectively:

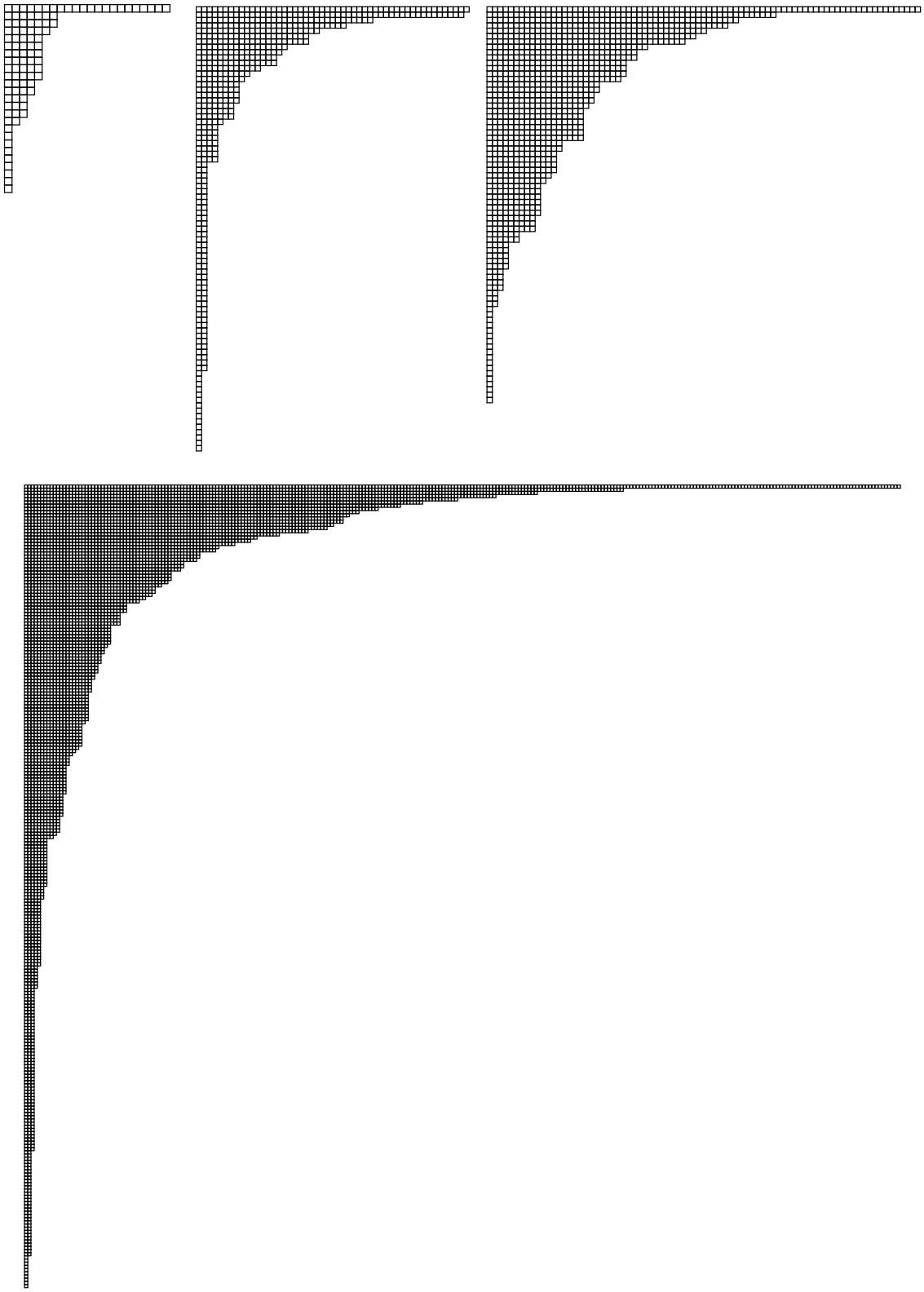

Figure 2.5: Random partitions of $n = 100$, 500, 1000, and 5000.

2.3 Set Partitions

Definition 2.3 (Set Partitions)**.** A *partition* of a finite set S is a collection $\{X_1, X_2, ..., X_k\}$ of subsets of S such that $\bigcup_{i=1}^{k} X_i = S$ and $X_i \cap X_j = \emptyset$ for all $i \neq j$. We call the sets X_i *blocks* of the partition.

For example, the 5 partitions of the set $S = \{1, 2, 3\}$ are: $\{1, 2, 3\}$, $\{\{1, 2\}, \{3\}\}$, $\{\{1, 3\}, \{2\}\}$, $\{\{1\}, \{2, 3\}\}$, and $\{\{1\}, \{2\}, \{3\}\}$. We can illustrate these partitions using Young diagrams where each row demonstrates the size of a block. The elements of each block are placed in the boxes in increasing order. These diagrams are known as *tableaux* and are shown for the above example of $S = \{1, 2, 3\}$ in Figure 2.6:

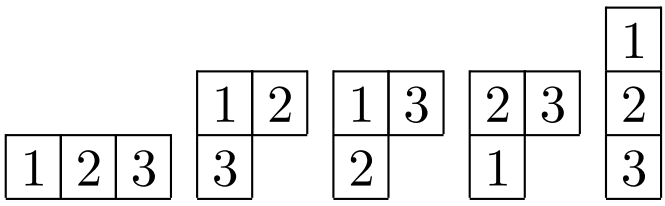

Figure 2.6: Tableaux for partitions of {1, 2, 3}.

The size of the partition is the sum of the block sizes ($\sum_{i=1}^{k} |X_i| = |S|$). Elements from the set are the atomic objects and we allow for the unique partition ϵ of the empty set, which is of size 0 and contains 0 blocks. Let $\mathscr{S}$ be the class of all set partitions. Since elements in the set are distinct, it is evident that $\mathscr{S}$ is a labelled class.

We can form a partition of $S = \{1, 2, ..., n\}$ by first fixing element n to be in a particular block. There are k elements not in that block, for some integer $k \in [0, n-1]$. There are $\binom{n-1}{k}$ ways to choose these elements, and then s_k ways to partition them. To account for all possible values of k, we obtain the recursion for the famous *Bell numbers*:

$$s_n = \sum_{k=0}^{n-1} \binom{n-1}{k} s_k$$

with initial condition $s_0 = 1$. Now we divide both sides by s_n to obtain a probability distribution

$$1 = \sum_{k=0}^{n-1} \binom{n-1}{k} \frac{s_k}{s_n}.$$

Interpreting this decomposition, we can develop a recursive method sampler.

Example 2.4 (Recursive Method Sampler for Set Partitions)**.** A pseudo-code implementation is displayed in Algorithm 2.3. The function chooseBlocks($m, blocks$) is responsible for choosing the sizes of each of the blocks in the partition. This is where the probability distribution and recursion are implemented. First we assign x a random integer between 1 and s_n and we initialize k as 1 (since blocks must be nonempty) and *tally* as 0. The probability distribution is then implemented in lines 9 through 13. For each integer $i \in [0, m]$ we increase our tally by the next value of the summand $(\binom{m-1}{k-1} s_{m-k}$, since k starts at 1). We increment k at the end of each iteration and break from the loop once the tally exceeds the value chosen for x. We then append k to our list of block sizes and recurse on $m - k$, once $m = 0$ we return the list of block sizes. The function genSetPart(n) is then responsible for using chooseBlocks($m, blocks$) to create the random set partition. We return $\{\epsilon\}$ if $|S| = 0$ and otherwise we first select our block sizes by calling chooseBlocks with parameters n and an empty list for the block sizes. Next we call on a function that can generate a random permutation of S. Since the blocks are stored in a list and we place elements into them sequentially, the random permutation allows each element to have equal probability of ending up in any block. Consider the example $S = \{1, 2, 3, 4, 5\}$ where we have selected blocks of size 3 and 2 respectively. If we place the elements into the blocks without the permutation then we would always have elements 1, 2, and 3 ending up in the block of size 3 together and elements 4 and 5 in the block of size 2. We omit the pseudo-code for random permutations in Algorithm 2.3 as there are many well known simple ways of doing this, however it is included in the implementation

in Appendix A.3. We then initialize our variables *part* (as an empty list) and *pos* (as 0) and then in lines 23 throgh 31 we build the partition. We iterate over each of the values in *blocks*, first creating a *temp* variable to hold the values assigned to that block. For each integer $j \in [0, blocks_i]$ we append S_{j+pos} to our *temp* list. We then append *temp* to our partition and increment *pos* by the size of the previous block. Finally, we return our partition. The implementation of Algorithm 2.3 is displayed in Appendix A.3.

Algorithm 2.3: Recursive method set partition sampler.

Data: n - the cardinality of the set to partition
Result: a randomly generated partition of $\{1, 2, ..., n\}$

Compute Bell numbers up to any including the n^{th} term and store in array s.
$S \leftarrow \{1, 2, ..., n\}$

function chooseBlocks($m, blocks$):
  if $m = 0$ **then**
    return *blocks*
  $x \leftarrow$ random integer between 1 and s_m
  $k \leftarrow 1$
  $tally \leftarrow 0$
  for i ***from*** 0 ***to*** m **do**
    $tally \leftarrow \binom{m-1}{k-1} s_{m-k}$
    if $x \leq tally$ **then**
      break
    $k \leftarrow k + 1$
  append k to *blocks*
  return chooseBlocks($m - k$)

function genSetPart(n):
  if $n = 0$ **then**
    return ϵ
  $blocks \leftarrow$ chooseBlocks($n, [\,]$)
  randPermutation(S)
  $part \leftarrow \{\}$
  $pos \leftarrow 0$
  for i ***from*** 0 ***to*** *length(blocks)* **do**
    $temp \leftarrow \{\}$
    for j ***from*** 0 ***to*** $blocks[i]$ **do**
      append S_{j+pos} to *temp*
    append *temp* to *part*
    $pos \leftarrow pos + blocks[i]$
  return *part*

Illustrated in Figure 2.7 are random set partitions where $|S| = 50$, 100, and 200 respectively.

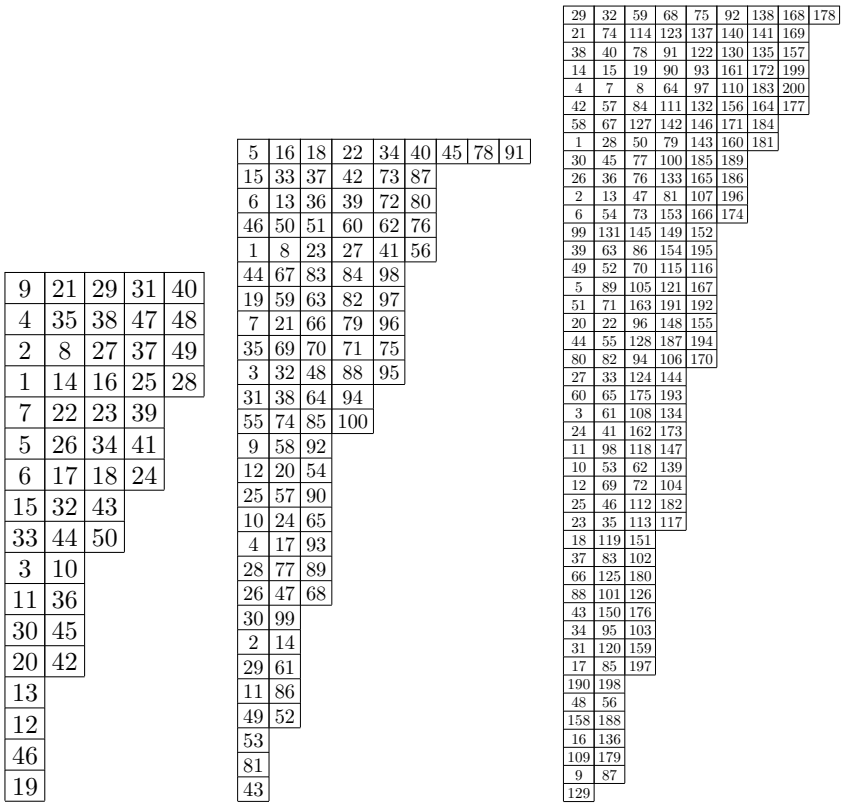

Figure 2.7: Random partitions of sets of cardinality 50, 100, and 200.

3 Boltzmann Samplers

The Boltzmann sampling method derives its name from Boltzmann distributions which give the probability of a system existing in a certain state as a function of the temperature of the system and the energy of the state [4]. Unlike recursive method samplers, the Boltzmann sampling method does not require computing the table of $\{c_k\}$ values, nor does it require a recursive decomposition for the class being considered. Boltzmann samplers require a closed form for the generating function of $\mathscr{C}$. The second major difference is that Boltzmann samplers are not guaranteed to return an object of the desired size on any given call. This of course sounds counter-intuitive as this means in order to attain an object of size n, multiple runs of the function would almost inevitably be required.

Boltzmann samplers rely on calculating a continuous control parameter, $x > 0$ and in putting a measure over the whole of $\mathscr{C}$. The objective is to tune x in order to maximize the probability of obtaining some $\gamma \in \mathscr{C}$ such that $\gamma \in \mathscr{C}_n$, for the desired size n. With the Boltzmann model, we sample over the whole of $\mathscr{C}$ in such a way that the object γ that is generated is selected uniformly from all objects of the same size. This introduces the possibility of two sampling strategies. First we have *exact size sampling* wherein we condition the sampler to only return $\gamma \in \mathscr{C}_n$. Second we have *approximate size sampling* wherein we condition the sampler to only return $\gamma \in \mathscr{C}_N$, where $(1-\varepsilon)n \leq N \leq (1+\varepsilon)n$, for some predetermined $\varepsilon > 0$.

It is evident that Boltzmann samplers, in general, fall short of their recursive method counterparts with respect to time efficiency. However Boltzmann samplers offer a space efficient alternative that permit effective sampling of objects of sizes that are out of reach for the recursive method. We begin by discussing the calculus behind Boltzmann samplers.

3.1 The Calculus of Boltzmann Samplers

Under the Boltzmann model we aim to select a control parameter $x > 0$ that maximizes the probability of generating an object of the desired size. Boltzmann models come in two varieties, one used in the ordinary (unlabelled) case and another used in the exponential (labelled) case. Given the OGF and EGF for $\mathscr{C}$ specified in Section 1.1 the probability of obtaining any $\gamma \in \mathscr{C}$ is

$$\text{ordinary/unlabelled case:} \quad \mathbb{P}_x(\gamma) = \frac{x^{|\gamma|}}{C(x)} \ , \ C(x) = \sum_{\gamma \in \mathscr{C}} x^{|\gamma|}$$

$$\text{exponential/labelled case:} \quad \mathbb{P}_x(\gamma) = \frac{x^{|\gamma|}}{\widehat{C}(x)|\gamma|!} \ , \ \widehat{C}(x) = \sum_{\gamma \in \mathscr{C}} \frac{x^{|\gamma|}}{|\gamma|!}$$

Definition 3.1 (Boltzmann Samplers)**.** A *Boltzmann sampler* is a a process that

produces objects from a class $\mathscr{C}$ in accordance with the appropriate Boltzmann model (ordinary or exponential) [4]. We denote the Boltzmann sampler for $\mathscr{C}$ with parameter x by $\Gamma C(x)$.

As mentioned, Boltzmann samplers do not guarantee that an object of the desired size will be returned on any given call and so we tune x so that the probability of obtaining an object of the desired size (or within the desired range of sizes) is reasonably high [4, 3]. The probabilities for the ordinary (unlabelled) and exponential (labelled) cases of getting an object of exactly size n are given respectively by the following

$$\mathbb{P}_x(N=n) = \frac{c_n x^n}{C(x)} \quad \text{or} \quad \mathbb{P}_x(N=n) = \frac{c_n x^n}{n!\widehat{C}(x)},$$
$$\text{where } C(x) = \sum_n c_n x^n \quad \text{and} \quad \widehat{C}(x) = \sum_n \frac{c_n x^n}{n!}$$

where permissible values for $x > 0$ are all values such that $C(x)$ (or $\widehat{C}(x)$) is defined.

We now consider how to tune the parameter x in order to maximize $\mathbb{P}_x(N=n)$.

Theorem 3.1 (Moments of the Random Variable N)**.** *The random size of the object produced under the ordinary (unlabelled) Boltzmann model of parameter x has first and second moments satisfying*

$$\mathbb{E}_x(N) = \frac{xC'(x)}{C(x)} \quad \text{and} \quad \mathbb{E}_x(N^2) = \frac{x^2C''(x) + xC'(x)}{C(x)}.$$

Let $\mathbb{V}$ denote the variance operator, then

$$\mathbb{V}_x(N) = x\frac{d}{dx}\mathbb{E}_x(N).$$

Proof. If we consider the Boltzmann model for the ordinary case then the probability

generating function of N is

$$\sum_n \mathbb{P}_x(N=n)z^n = \sum_n \frac{c_n x^n}{C(x)} z^n = \frac{C(xz)}{C(x)}$$

since $\mathbb{P}_x(N=n) = \frac{c_n x^n}{C(x)}$. We then make use of the generating function given above for the ordinary, unlabelled case. Since derivatives tell us about how a function changes and our control parameter x is always continuous over some domain, we can differentiate setting $z = 1$:

$$\begin{aligned}
\mathbb{E}_x(N) &= \left(\frac{\partial}{\partial z} \frac{C(xz)}{C(x)} \right)_{z=1} \\
&= \frac{xC'(x)}{C(x)}
\end{aligned}$$

$$\begin{aligned}
\mathbb{E}_x(N(N-1)) &= \left(\frac{\partial^2}{\partial z^2} \frac{C(xz)}{C(x)} \right)_{z=1} \\
\mathbb{E}_x(N^2 - N) &= \frac{x^2 C''(x)}{C(x)} \\
\mathbb{E}_x(N^2) &= \frac{x^2 C''(x)}{C(x)} + \frac{xC'(x)}{C(x)} \\
&= \frac{x^2 C''(x) + xC'(x)}{C(x)}.
\end{aligned}$$

Now we consider the formula for variance

$$\mathbb{V}_x(N) = \mathbb{E}_x[(N-\mu)^2].$$

Since $\mathbb{E}_x(N)$ is the first moment of the random variable N, we have $\mu = \mathbb{E}_x(N)$.

Therefore

$$\begin{aligned}
\mathbb{V}_x(N) &= \mathbb{E}_x[(N - \mathbb{E}_x(N))^2] \\
&= \mathbb{E}_x[N^2 - 2N\mathbb{E}_x(N) + (\mathbb{E}_x(N))^2] \\
&= \mathbb{E}_x(N^2) - 2\mathbb{E}_x(N)\mathbb{E}_x(N) + (\mathbb{E}_x(N))^2 \\
&= \mathbb{E}_x(N^2) - 2(\mathbb{E}_x(N))^2 + (\mathbb{E}_x(N))^2 \\
&= \mathbb{E}_x(N^2) - (\mathbb{E}_x(N))^2 \\
&= \frac{x^2C''(x) + xC'(x)}{C(x)} - \left(\frac{xC'(x)}{C(x)}\right)^2.
\end{aligned}$$

Now we differentiate $\mathbb{E}_x(N)$ and multiply by x and note that we obtain the same result

$$\begin{aligned}
x\frac{d}{dx}\mathbb{E}_x(N) &= x\frac{d}{dx}\left(\frac{xC'(x)}{C(x)}\right) \\
&= x\left(\frac{C(x)(C'(x) + xC''(x)) - C'(x)(x(C'(x))}{C(x)^2}\right) \\
&= \frac{x^2C''(x) + xC(x)}{C(x)} - \left(\frac{xC'(x)}{C(x)}\right)^2 \\
&= \mathbb{V}_x(N).
\end{aligned}$$

Replacing $C(x)$ with $\widehat{C}(x)$ gives the first and second moments satisfied in the exponential case [4]. □

Using Theorem 3.1 we can effectively tune the parameter x in order to maximize the probability of generating an object of size n by solving $\mathbb{E}_x(N) = n$ for x.

The calculus behind Boltzmann samplers allows us to build up samplers from simpler classes of objects to sample from more complex classes. This is done through a variety of constructs including finite sets, the disjoint union, the cartesian product, sequences, and set classes [4]. If $\mathscr{C}$ can be expressed in terms of simpler classes of objects, say $\mathscr{A}$ and $\mathscr{B}$, using the constructs discussed later in this section, and we

have Boltzmann samplers $\Gamma A(x)$ and $\Gamma B(x)$ for $\mathscr{A}$ and $\mathscr{B}$ respectively then we can create from them a new Boltzmann sampler which is equivalent to $\Gamma C(x)$ [4].

3.1.1 Probability Distributions and Random Variables

Definition 3.2 (Bernoulli Distribution)**.** Let $0 \leq p \leq 1$. A *Bernoulli distribution* with parameter p is a discrete probability distribution of a random variable that takes the value 1 with probability p or 0 with probability $q = 1 - p$. Let Bern(p) denote a generator for a Bernoulli variable with parameter p. That is,

$$\text{Bern}(p) = 1 \text{ with probability } p \text{ and } \text{Bern}(p) = 0 \text{ with probability } q = 1 - p.$$

Note that this is a Boolean variable with 1 corresponding to $true$ and 0 corresponding to $false$. A *Bernoulli trial* refers to a random experiment having two possible outcomes: success or failure.

Definition 3.3 (Geometric Distribution)**.** A *geometric variable* with parameter p counts the number of successes before a failure in a sequence of repeated independent Bernoulli trials, each equipped with parameter p. The corresponding *geometric distribution* is then the distribution of this variable. The probability that k successes occur before a failure is

$$\mathbb{P}(X = k) = p^k(1 - p)$$

Let p_k be the probability that a random variable with the geometric distribution has value k. Let Geom(p) denote a generator for a geometric random variable with parameter p. Then

$$\text{Geom}(p) = k \text{ with probability } p_k = (1 - p)p^k.$$

Definition 3.4 (Poisson Distribution)**.** A *Poisson distribution* is a discrete proba-

bility distribution that expresses the probability that a particular number of events occur in a set interval of space or time. The events must occur at a known, constant rate and their occurrence is independent of all previous events. Let λ denote the average number of events that occur in an interval. Then the probability that k events occur in an interval is given by

$$\mathbb{P}(X = k) = e^{-\lambda}\frac{\lambda^k}{k!}.$$

Let p_k be the probability that a random variable with the Poisson distribution has value k. Let Pois(λ) denote a generator for a Poisson random variable with parameter λ. Then

$$\text{Pois}(\lambda) = k \text{ with probability } p_k = e^{-\lambda}\frac{\lambda^k}{k!}.$$

3.1.2 Finite Sets

We can sample from a finite set $\mathscr{F} = \{z_1, z_2, z_3, ..., z_n\}$ by selecting the i^{th} element of $\mathscr{F}$ with probability

$$\mathbb{P}(z_i) = \frac{x^{|z_i|}}{F(x)}, \text{ where } F(x) = \sum_i x^{|z_i|}$$

This is key as drawing from singleton classes is deterministic and we can form objects by considering two singleton classes, the one element of size 0 and the atomic object of a class, denoted $\mathscr{Z}$, from which all larger, more complex objects can be formed [4]. For example, in the case of binary trees the 0 element is a leaf and the atomic element is an internal node.

3.1.3 Disjoint Unions

Let $\mathscr{C} = \mathscr{A} + \mathscr{B}$. Let $\Gamma A(x)$, $\Gamma B(x)$ be the Boltzmann samplers for $\mathscr{A}$ and $\mathscr{B}$ respectively. We can implement a Bernoulli distribution to sample from $\mathscr{C}$ by using

the samplers for $\mathscr{A}$ and $\mathscr{B}$. By definition, the probability of obtaining some $\alpha \in \mathscr{A}$ from the sampler $\Gamma C(x)$ is

$$\mathbb{P}_{\mathscr{C},x}(\alpha) \equiv \frac{x^{|\alpha|}}{C(x)} = \frac{x^{|\alpha|}}{A(x)} \cdot \frac{A(x)}{C(x)}$$

and similarly for the probability of obtaining some $\beta \in \mathscr{B}$. It therefore follows that the overall probabilities of a randomly sampled $\gamma \in \mathscr{C}$ being from $\mathscr{A}$ or $\mathscr{B}$ are

$$\mathbb{P}_{\mathscr{C},x}(\gamma \in \mathscr{A}) = \frac{A(x)}{C(x)} \quad \text{and} \quad \mathbb{P}_{\mathscr{C},x}(\gamma \in \mathscr{B}) = \frac{B(x)}{C(x)}.$$

Let p_A be the probability that a randomly sampled $\gamma \in \mathscr{C}$ is from $\mathscr{A}$. We use the Bernoulli generator $\text{Bern}(p_A)$ to develop the Boltzmann sampler $\Gamma C(x)$ which generates objects from $\mathscr{C} = \mathscr{A} + \mathscr{B}$. The pseudo-code is given in Algorithm 3.1.

Algorithm 3.1: Boltzmann disjoint union constructor.

```
1 function ΓC(x):
2 p_A ← A(x)/(A(x)+B(x))
3 if Bern(p_A) then
4 |   return ΓA(x)
5 else
6 |   return ΓB(x)
```

We abbreviate this as

$$\left(\text{Bern}\left(\frac{A(x)}{C(x)}\right) \longrightarrow \Gamma A(x) | \Gamma B(x)\right).$$

3.1.4 Cartesian Products

Let $\mathscr{C} = \mathscr{A} \times \mathscr{B}$. If we have Boltzmann samplers $\Gamma A(x)$ and $\Gamma B(x)$ for $\mathscr{A}$ and $\mathscr{B}$ respectively then we can create the sampler for $\mathscr{C}$ [4]. Since $\mathscr{C} = \mathscr{A} \times \mathscr{B}$ we have by

Lemma 1.3 that the generating function of $\mathscr{C}$ is

$$C(z) = A(z) \cdot B(z) = \sum_{(\alpha,\beta)\in\mathscr{A}\times\mathscr{B}} z^{|\alpha|+|\beta|}.$$

As such the probability of obtaining any given $\gamma \in \mathscr{C}$ where $\gamma = (\alpha, \beta)$ is

$$\mathbb{P}_{\mathscr{C},x}(\gamma) \equiv \frac{x^{|\gamma|}}{C(x)} = \frac{x^{|\alpha|}}{A(x)} \cdot \frac{x^{|\beta|}}{B(x)}$$

and so we can construct the sampler $\Gamma C(x)$ for $\mathscr{C}$ by drawing independently from the samplers $\Gamma A(x)$ and $\Gamma B(x)$ to obtain an ordered pair $(\alpha, \beta) = \gamma \in \mathscr{C}$ which is generated uniformly at random [4]. The sampler for $\mathscr{C}$ is demonstrated in Algorithm 3.2.

Algorithm 3.2: Boltzmann Cartesian product constructor.

```
1 function ΓC(x) :
2 return (ΓA(x), ΓB(x))
```

We abbreviate this as $\Gamma C(x) = (\Gamma A(x); \Gamma B(x))$. We can also use the natural extension $(f_1; f_2; ...; f_r)$ where r-tuples are involved [4].

3.1.5 Sequences

We write $\mathscr{C} = \mathfrak{S}(\mathscr{A})$ to denote that $\mathscr{C}$ consists of all finite lists of elements from $\mathscr{A}$ ($\mathscr{C} = \{\epsilon\} + \mathscr{A} + \mathscr{A}^2 + ...$). We deduce that $\mathscr{C} = \{\epsilon\} + \mathscr{A} \times \mathscr{C}$ [4]. Hence by Lemmas 1.1 and 1.3 we have $C(x) = 1 + A(x)C(x)$ and thus

$$C(x) = \frac{1}{1 - A(x)}.$$

From here we can develop the sampler $\Gamma C(x)$ either through a recursive design which relies on Bernoulli variables or a geometric design that calls for the use of Geometric variables. The pseudo-code for the recursive sampler is given in Algorithm 3.3 and

the pseudo-code for the geometric sampler is shown in Algorithm 3.4.

Algorithm 3.3: Boltzmann recursive sequence sampler.

```
function ΓC(x):
if Bern(A(x)) then
    return (ΓA(x); ΓC(x))
else
    return ϵ
```

Algorithm 3.3 works because $\Gamma C(x)$ is built via disjoint union, hence we get a Bernoulli switch with

$$p = \frac{A(x)C(x)}{1 + A(x)C(x)} = \frac{A(x)C(x)}{C(x)} = A(x).$$

Algorithm 3.4: Boltzmann geometric sequence sampler.

```
function ΓC(x):
draw k based on Geom(A(x))
return k-tuple given by (ΓA(x); ΓA(x); ...; ΓA(x))
```

Algorithm 3.4 is effectively a simplification of Algorithm 3.3 since instead of repeatedly drawing from a Bernoulli random variable until a result of 0 is attained, we simply use a geometric random variable with rate $A(x)$ [4]. We abbreviate this as $\Gamma C(x) = (\text{Geom}(A(x)) \Longrightarrow \Gamma(x)$.

3.1.6 Set Class

This construct applies only to the labelled case. The class $\mathscr{C}$ is the set-class of another class, $\mathscr{A}$, denoted $\mathscr{C} = \mathfrak{P}(\mathscr{A})$, if $\mathscr{C}$ is the quotient of sequences $\mathfrak{S}(\mathscr{A})/\equiv$, where $\equiv$ is the relation that deems two sequences to be equivalent if one derives from the other via an arbitrary permutation of components [4]. Now let $\widehat{A}(x)$ be the EGF of $\mathscr{A}$, and note that the EGF of $\mathscr{C}$ is

$$\widehat{C}(x) = \sum_{k=0}^{\infty} \frac{1}{k!} \widehat{A}(x)^k = e^{\widehat{A}(x)}.$$

Consider the exponential Boltzmann model. The probability that a set $\mathscr{C}$ has k components of $\mathscr{A}$ is

$$\frac{1}{\widehat{C}(x)}\frac{1}{k!}\widehat{A}(x)^k = e^{-\widehat{A}(x)}\frac{\widehat{A}(x)^k}{k!}$$

which is equivalent to a Poisson law of rate $\widehat{A}(x)$. Hence we can devise a simple algorithm for generating sets, the pseudo-code is given in Algorithm 3.5.

Algorithm 3.5: Boltzmann set class constructor.

```
1 function ΓC(x):
2 return Pois(Â(x))
```

We abbreviate this as $\Gamma C(x) = \text{Pois}(\widehat{A}(x)) \Longrightarrow \Gamma A(x)$.

3.2 Binary Trees

To create a Boltzmann sampler for binary trees we first consider Defintion 2.1 and recall the decomposition $\mathscr{B} = \{\circ\} + (\{\bullet\} \times \mathscr{B}^2)$. We can determine the parameter x that should be used in order to maximize the probability of attaining trees of size n using the closed form OGF given in Section 2.1 in combination with Theorem 3.1:

$$\begin{aligned} B(x) &= \frac{1-\sqrt{1-4x}}{2} \\ B'(x) &= \frac{1}{\sqrt{1-4x}} \\ \mathbb{E}_x(N) &= \frac{xB'(x)}{B(x)} = \frac{2x}{4x+\sqrt{1-4x}-1}. \end{aligned}$$

Setting $\mathbb{E}_x(N) = n$ and solving for x, we obtain

$$n = \frac{2x}{4x+\sqrt{1-4x}-1} \Longrightarrow x = \frac{n(n-1)}{(2n-1)^2}$$

for $n \geq 0$.

We modify the decomposition to be $\mathscr{B} = \mathscr{Z} + (\mathscr{B} \times \mathscr{B})$, where $\mathscr{Z}$ is the class

comprising of the generic node and so we count only with respect to leaves. We can do this as the sampler $\Gamma Z(x)$ is deterministic and consists only of the instruction to generate a node. Thus the Boltzmann sampler then follows from the Cartesian product and union constructs:

$$\Gamma B(x) = (\text{Bern}(p_0) \to \mathscr{Z} | (\Gamma B(x); \Gamma B(x)))$$
$$\text{where } p_0 = \frac{x}{B(x)} = \frac{x}{\frac{1-\sqrt{1-4x}}{2}} = \frac{2x}{1-\sqrt{1-4x}}.$$

That is, with probability p_0 we terminate at a leaf node and with probability $1 - p_0$ we sample from the Cartesian product $(\Gamma B(x); \Gamma B(x))$. We demonstrate a pseudo-code implementation of the sampler $\Gamma B(x)$ in Algorithm 3.6.

Algorithm 3.6: Boltzmann sampler for binary trees.

Data: n - the desired number of internal nodes
Data: $x \leftarrow \frac{n(n-1)}{(2n-1)^2}$
function $\Gamma B(x)$:
  $p_0 \leftarrow \frac{2x}{1-\sqrt{1-4x}}$
  $u \leftarrow rand(0, 1)$
  if $u < p_0$ **then**
    return $\mathscr{Z}$
  else
    return $(\Gamma B(x), \Gamma B(x))$

The implementation of $\Gamma B(x)$ (Algorithm 3.6) is displayed in Appendix B.1.

It must be noted however that the above sampler has an extraordinarily poor hit rate for large n and as such is undesirable for practical use. Indeed we can observe based on Theorem 3.1 that the theoretical variance is

$$\begin{aligned}
\mathbb{V}_x(N) &= x\frac{d}{dx}\mathbb{E}_x(N) \\
&= x\frac{d}{dx}\left(\frac{2x}{4x+\sqrt{1-4x}-1}\right) \\
&= \frac{x}{(1-4x)^{\frac{3}{2}}}.
\end{aligned}$$

At the optimal value, $x = \frac{n(n-1)}{(2n-1)^2}$, this gives

$$\begin{aligned}\mathbb{V}_x(N) &= \frac{\frac{n(n-1)}{(2n-1)^2}}{\left(1 - \frac{4n(n-1)}{(2n-1)^2}\right)^{\frac{3}{2}}} \\ &= n(n-1)|1-2n| \\ &= (n^2-n)(2n-1) \text{ (since } n \geq 1)\end{aligned}$$

hence

$$\sigma = \sqrt{(n^2-n)(2n-1)}.$$

For example, for $n = 500$ we get $x = 0.2499997495$ and $\mathbb{V}_x(N) = 249250500$, and consequenty $\sigma = 15787.669$, a standard deviation that exceeds 31 times the expected mean. Indeed, we can see that $\sigma \sim \sqrt{2n^3}$.

3.2.1 Pointed Binary Trees

In order to circumvent the above downfall of the Boltzmann sampler for binary trees the notion of the *pointing* operation becomes key. In the standard implementaion outlined above, a binary tree is thought of in a similar manner as in Section 2.1 where

$$\mathscr{B} = \mathscr{Z} + (\mathscr{B} \times \mathscr{B}).$$

However as noted the implementation that results has severe performance deficiencies.

Definition 3.5 (The Pointing Operator)**.** Given a class $\mathscr{C}$ we refer to $\mathscr{C}^\bullet = \{(\gamma, i) : \gamma \in \mathscr{C}, i \in \{1, 2, ..., |\gamma|\}\}$ or equivalenty ${\mathscr{C}^\bullet}_n \cong \mathscr{C}_n \times \{1, 2, ..., n\}$ as the class of *pointed* objects from $\mathscr{C}$ [4]. Objects in $\mathscr{C}^\bullet$ are effectively objects from $\mathscr{C}$ where we distinguish one of the atoms by *pointing* to it.

From Definition 3.5 we have $|\mathscr{C}_n| = n|\mathscr{C}|$ and hence the generating function for $\mathscr{C}^\bullet$ is $C^\bullet(z) = z\frac{d}{dz}C(z)$. Let $\mathscr{B}^\bullet$ be the class of pointed binary trees, the OGF of $\mathscr{B}^\bullet$

is

$$B^\bullet(x) = x\frac{d}{dx}B(x) = \frac{x}{\sqrt{1-4x}}.$$

Thus we obtain a new decomposition for $\mathscr{B}^\bullet$

$$\mathscr{B}^\bullet = \mathscr{Z}^\bullet + (\mathscr{B}^\bullet \times \mathscr{B}) + (\mathscr{B} \times \mathscr{B}^\bullet).$$

We again make use of the cross product and disjoint union constructs to develop a new sampler, $\Gamma B^\bullet(x)$ which builds upon the original sampler $\Gamma B(x)$ in the following way. First we determine $\mathbb{E}_x(N)$

$$\begin{aligned} B^\bullet(x) &= \frac{x}{\sqrt{1-4x}} \\ B^{\bullet'}(x) &= \frac{1-2x}{(1-4x)^{\frac{3}{2}}} \\ \mathbb{E}_x(N) &= \frac{xB^{\bullet'}(x)}{B^\bullet(x)} \\ &= \frac{1-2x}{1-4x}. \end{aligned}$$

Setting $\mathbb{E}_x(N) = n$ and solving for x we get

$$n = \frac{1-2x}{1-4x} \Longrightarrow x = \frac{n-1}{4n-2}.$$

Then we construct $\Gamma B^\bullet(x)$ as follows:

$$\Gamma B(x) = (\text{Bern}(p_0) \to \mathscr{Z}|(\Gamma B(x); \Gamma B(x)))$$
$$\Gamma B^\bullet(x) = (\text{Bern}(p_1, p_2) \to \mathscr{Z}^\bullet|(\Gamma B^\bullet(x); \Gamma B(x))|(\Gamma B(x); \Gamma B^\bullet(x)))$$

where p_0 is as before,

$$p_1 = \sqrt{1-4x}\ , \quad p_2 = \frac{1}{2} - \frac{1}{2}\sqrt{1-4x},$$

and Bern(p_1, p_2) is a probabilistic switch where we terminate with a pointed leaf with probability p_1, sample from $(\Gamma B^\bullet(x); \Gamma B(x))$ with probability $p_1 + p_2$, or sample from $(\Gamma B(x); \Gamma B^\bullet(x))$ with probability $1 - (p_1 + p_2)$. Taking note of the fact that $p_1 \approx 0$ for $x \approx \frac{1}{4}$, it is evident that $\Gamma B^\bullet(x)$ will produce a terminal node with extremely low probability, and with high probability it will generate a sequence of calls to itself and to $\Gamma B(x)$ which itself produces either calls to itself or a terminal node each with probability close to $\frac{1}{2}$. As a result $\Gamma B^\bullet$ effectively controls for the issue of generating an excessive quantity of small trees. The implementation of $\Gamma B^\bullet$ is displayed in Algorithm 3.7.

Algorithm 3.7: Boltzmann sampler for pointed binary trees

Data: n - the desired number of internal nodes
Data: $x \leftarrow \frac{n-1}{4n-2}$
function $\Gamma B^\bullet(x)$:
  $p_1 \leftarrow \sqrt{1-4x}$
  $p_2 \leftarrow \frac{1}{2} - \frac{1}{2}\sqrt{1-4x}$
  $u \leftarrow rand(0,1)$
  if $u < p_1$ **then**
    return $\mathscr{Z}^\bullet$
  if $p_1 \leq u < p_1 + p_2$ **then**
    return $(\Gamma B^\bullet(x), \Gamma B(x))$
  else
    return $(\Gamma B(x), \Gamma B^\bullet(x))$

The implementation of $\Gamma B^\bullet$ (Algorithm 3.7) is displayed in Appendix B.1.

While this does not yield a perfect sampler, the pointing mechanism ensures significantly higher probabilities of obtaining larger trees and moreoever, a greater probability of obtaining trees in the vicinity of the desired size. By Theorem 3.1 the

theoretical variance and standard deviation with the pointing operator are:

$$\begin{aligned}\mathbb{V}_x(N) &= x\frac{d}{dx}\mathbb{E}_x(N)\\ &= x\frac{d}{dx}\left(\frac{1-2x}{1-4x}\right)\\ &= \frac{2x}{(1-4x)^2}.\end{aligned}$$

At the optimal value, $x = \frac{n-1}{4n-2}$, this gives

$$\begin{aligned}\mathbb{V}_x(N) &= \frac{\frac{2(n-1)}{4n-2}}{\left(1-\frac{4(n-1)}{4n-2}\right)^2}\\ &= (n-1)(2n-1)\end{aligned}$$

hence

$$\sigma = \sqrt{(n-1)(2n-1)}.$$

For example, for $n = 500$ we get that $x = 0.2497497497$ and that $\mathbb{V}_x(N) = 4985$ and consequenty $\sigma = 706.046$, a significant improvement over the non-pointed sampler [4]. Indeed we now see that $\sigma \sim \sqrt{2}n$.

To further illustrate by the example with $n = 500$ and the aforementioned x values we get the following size trees on a run of 25 samples for each:

$$\Gamma B(x) \to \{2, 1, 1, 11, 5, 1, 2, 1, 19, 56, 1, 9, 1, 1, 1, 2, 2, 1, 3, 1, 1, 4, 33, 1, 1\}$$

$$\Gamma B^\bullet(x) \to \{6, 1580, 181, 11, 135, 3178, 1139, 4, 74, 1872, 7, 698, 1671, 118, 54, 778,$$

$$120, 97, 16, 126, 112, 199, 364, 8, 1\}$$

cementing the utility of the pointing mechanism.

3.3 Integer Partitions

In order to design a Boltzmann sampler for integer partitions we must first consider Definition 2.2 recalling that integer partitions are lists of integers. Since λ is a weakly increasing list of integers, we ensure that each partition is obtained in only one way. As with the recursive method implementation, there are some unique properties of integer partitions that must be carefully considered. Since Boltzmann samplers are not required, on any given iteration, to produce an object of the exact size desired, we have that partitions greater than or less than n may be produced (for any $n \geq 1$). In order to ensure that the sampler will produce an output, as with the recursive method we will only consider partitions with parts at most n if we desire a partition of n. This then results in a sampler that will produce uniformly random integer partitions with parts of at most n. In the case that the output partition is not larger than n, this will be a partition chosen uniformly at random from all possibilities [2].

From the above definitions it follows that an integer partition is the Cartesian product of sequences of integers $i \in [1, n]$. Let $\mathscr{P}_n$ denote the subclass of all partitions with parts at most n, then

$$\mathscr{P}_n = \mathfrak{S}(\mathscr{Z}) \times \mathfrak{S}(\mathscr{Z}^2) \times \cdots \times \mathfrak{S}(\mathscr{Z}^n).$$

Thus the corresponding Boltzmann sampler is the Cartesian product of sequence samplers [2, 4]. For each $\mathscr{Z}^k$ we have OGF x^k and hence probability $p = x^k$ for each $\mathfrak{S}(\mathscr{Z}^k)$. Since the desired result is to have the Boltzmann sampler achieve partitions of either exactly n or approximately n, we must make use of Theorem 3.1 again and thus we note that the OGF for integer partitions with parts at most n is

$$P_n(x) := \sum_{m \geq 0} p(m, n) x^m = \prod_{i=1}^{n} \frac{1}{1 - x^i}.$$

Differentiation yields

$$P_n'(x) = P_n(x) \sum_{k=1}^{n} \frac{kx^{k-1}}{1-x^k},$$

and by Theorem 3.1 we get

$$\mathbb{E}_x(N) = \frac{xP_n'(x)}{P_n(x)} = x \sum_{k=1}^{n} \frac{kx^{k-1}}{1-x^k} = \sum_{k=1}^{n} \frac{kx^k}{1-x^k} = \sum_{k=1}^{n} \frac{k}{x^{-k}-1}.$$

In order to attain the greatest probability of the sampler producing partitions of n, we need to find $x \in (0,1)$ such that the sum given by $\mathbb{E}_x(N)$ is exactly n. However as n grows in size this task becomes increasingly out of reach even via the use of numerical computing. As a result we instead aim to find $x \in (0,1)$ such that $\mathbb{E}_x(N) \approx n$. To do this we will make use of the integral approximation of $\mathbb{E}_x(N)$:

$$\sum_{k=1}^{n} \frac{k}{x^{-k}-1} \approx \int_0^n \frac{z}{x^{-z}-1} dz.$$

In this integral the parameter x is constant and there is a singularity at $z = 0$, however we can apply l'Hopital's rule to show that the limit exists

$$\lim_{z \to 0} \frac{z}{x^{-z}-1} = \lim_{z \to 0} \frac{-1}{x^{-z} \ln x} = \frac{-1}{\ln x}.$$

Further note that $x \in (0,1)$ so $\ln x$ will be defined and $\ln x \neq 0$. We will now replace x^{-z} with $e^{-z \ln x}$ and use the substitution $u = -z \ln x$ to get

$$\int_0^n \frac{z}{x^{-z}-1} dz = \frac{1}{(\ln x)^2} \int_0^{-n \ln x} \frac{u}{e^u - 1} du \to \frac{1}{(\ln x)^2} \int_0^{\infty} \frac{u}{e^u - 1} du$$

as $u \to \infty$. Now note that the Riemann zeta function has integral representation

$$\zeta(s) = \frac{1}{\Gamma(s)} \int_0^{\infty} \frac{x^{s-1}}{e^x - 1} dx$$

where, if s is a positive integer, we have

$$\Gamma(s) = (s-1)!.$$

Now let $s = 2$ and note

$$\zeta(2) = \frac{1}{\Gamma(2)} \int_0^\infty \frac{x}{e^x - 1} dx = \frac{1}{1!} \int_0^\infty \frac{x}{e^x - 1} dx = \int_0^\infty \frac{x}{e^x - 1} dx.$$

Thus we obtain the same integrand as before and given the well known result

$$\zeta(2) = \frac{\pi^2}{6}$$

we have

$$\frac{1}{(\ln x)^2} \frac{\pi^2}{6} = n.$$

Thus we obtain as our solution

$$(\ln x)^2 = \frac{\pi^2}{6n}$$
$$\ln x = \pm \frac{\pi}{\sqrt{6n}}$$

and since we require $x \in (0, 1)$ we take the negative branch

$$\ln x = \frac{-\pi}{\sqrt{6n}}$$
$$x = e^{\frac{-\pi}{\sqrt{6n}}}.$$

With a feasibly computable equation now to obtain x we can now develop our sampler $\Gamma P(x)$. A pseudo-code implementation is shown in Algorithm 3.8.

Algorithm 3.8: Boltzmann sampler for integer partitions.

```
Data: n - the desired integer to partition
Data: x ← exp(−π/√(6n))
function ΓP(x) :
λ = [ ]
p ← 1
for i from 1 to n + 1 do
    p ← px
    do
        u ← rand(0, 1)
        if u < p then
            append i to λ
        else
            break
    while True;
return λ
```

The implementation of ΓP (Algorithm 3.8) is displayed in Appendix B.2.

Applying Theorem 3.1 to the integer partition sampler we obtain

$$\begin{aligned}\mathbb{V}_x(N) &= x\frac{d}{dx}\mathbb{E}_x(N)\\ &= x\frac{d}{dx}\sum_{k=1}^{n}\frac{k}{x^{-k}-1}\\ &= \sum_{k=1}^{n}\frac{k^2x^{-k}}{(x^{-k}-1)^2}.\end{aligned}$$

We see that $\mathbb{V}_x(N)$ is given by a computation that quickly becomes out of reach for large n, and so as before we approximate the sum via an integral

$$\begin{aligned}\mathbb{V}_x(N) &\approx \int_0^n \frac{z^2x^{-z}}{(x^{-z}-1)^2}dz\\ &= \int_0^n \frac{z^2e^{-zlnx}}{(e^{-zlnx}-1)^2}dz\\ &= -\frac{1}{(\ln x)^3}\int_0^{-nlnx}\frac{u^2e^u}{(e^u-1)^2}du.\end{aligned}$$

As before we have made the substitution $u = -z \ln x$. When $x = \alpha = \exp(\frac{-\pi}{\sqrt{6n}})$, and therefore $\ln x = \frac{-\pi}{\sqrt{6n}}$, we have

$$\mathbb{V}_\alpha(x) \approx \frac{(6n)^{\frac{3}{2}}}{\pi^3} \int_0^{\pi\sqrt{\frac{n}{6}}} \frac{u^2 e^u}{(e^u - 1)^2} du.$$

As $n \to \infty$

$$\int_0^{\pi\sqrt{\frac{n}{6}}} \frac{u^2 e^u}{(e^u - 1)^2} du \to \int_0^{\infty} \frac{u^2 e^u}{(e^u - 1)^2} du = \frac{\pi^2}{3}.$$

Thus we have an approximation for the variance when $x = \alpha$

$$\mathbb{V}_x(N) \approx \frac{2\sqrt{6}}{\pi} n^{\frac{3}{2}}$$

and so

$$\sigma \approx \sqrt{\frac{2\sqrt{6}}{\pi} n^{\frac{3}{2}}} = \left(\frac{24}{\pi^2}\right)^{\frac{1}{4}} n^{\frac{3}{4}}.$$

Using $n = 100$ as an example we get that $\mathbb{V}_\alpha(N) \approx 1559.39$ and consequently $\sigma \approx 39.099429$, noting that the true values are $\mathbb{V}_x(N) = 1528.77$ and $\sigma = 39.099488$. Running the sampler 1000 times for $n = 100$ we obtain $\mu = 93.727$, $\sigma^2 = 1386.5209$, and $\sigma = 37.236$.

3.4 Set Partitions

To develop a Boltzmann sampler for set partitions we first consider Definition 2.3 and note that a set partition is a set of sets. As a result it is evident that the use of the set class construct is required for such a sampler. Since set partitions are counted by the Bell numbers, we have that the closed form of the EGF for set partitions is

$$S(x) = \sum_{n=0}^{\infty} \frac{s_n x^n}{n!} = e^{e^x - 1}$$

Applying Theorem 3.1 we determine the appropriate value to choose for x

$$\mathbb{E}_x(N) = \frac{xS'(x)}{S(x)} = \frac{xe^{x+e^x-1}}{e^{e^x-1}} = xe^x.$$

Note that xe^x is the inverse of the Lambert W function, denoted $W(x)$, and thus we choose x such that $W(n) = x$.

We now apply the set class construct to develop the Boltzmann sampler for set partitions. First we define $\mathfrak{P}_{\geq 1}$ as the set class constructor with the additional constraint that there must be at least one component. As such we can define the set of all set partitions $\mathscr{S} = \mathfrak{P}(\mathfrak{P}_{\geq 1}(\mathscr{Z}))$, where the atom $\mathscr{Z}$ is labelled and is an element. We then require the use of two Poisson distributions; first one to select the number of blocks, k, which as a result the definition of set partitions will be Pois$(e^x - 1)$. We then select each of the block sizes, $b_1, b_2, ..., b_k$ such that each block size is independently selected at rate x from the Poisson law conditioned to accept only sizes of at least 1 and hence

$$\Gamma S(x) = \Big(Pois(e^x - 1) \Longrightarrow \Big(Pois_{\geq 1}(x) \Longrightarrow \mathscr{Z}\Big)\Big).$$

The pseudo-code implementation of the Poisson generators for $\Gamma S(x)$ are given in Algorithm 3.9.

Algorithm 3.9: Poisson generators for set partitions.

```
1  function Pois(x):
2  u ← rand(0, 1)
3  k ← 0
4  tally ← 0
5  do
6  |  if u < tally then
7  |  |  return k − 1
8  |  else
9  |  |  tally ← tally + e^{−x} x^k/k!
10 |  |  k ← k + 1
11 while True;

12 function Pois_{≥1}(x):
13 u ← rand(0, 1)
14 k ← 1
15 tally ← 0 do
16 |  if u < tally then
17 |  |  return k − 1
18 |  else
19 |  |  tally ← tally + 1/(e^x − 1) x^k/k!
20 |  |  k ← k + 1
21 while True;
```

Taking a random permutation of the set $S = \{1, 2, ..., n\}$, where $n = \sum_{i=1}^{k} b_i$, and placing the elements accordingly into each block will complete the process. The resulting output will be a partition of S that is randomly selected from all possible partitions of S. The pseudo-code for $\Gamma S(x)$ is shown in Algorithm 3.10.

Algorithm 3.10: Boltzmann sampler for set partitions

```
   Data: n - the desired cardinality of the partitioned set
   Data: x ← W(n)
 1 function ΓS(x) :
 2 k ← Pois(e^x − 1)
 3 n ← 0
 4 blocks ← [ ]
 5 for i from 0 to k do
 6 |   append Pois_{≥1}(x) to blocks
 7 |   n ← n + blocks[i]
 8 S ← {1, 2, ..., n}
 9 randPermutation(S)
10 part ← { }
11 pos ← 0
12 for i from 0 to k do
13 |   temp ← { }
14 |   for j from 0 to blocks[i] do
15 |   |   if i = 0 then
16 |   |   |   append S_j to temp
17 |   |   else
18 |   |   |   append S_{j+pos} to temp
19 |   append temp to part
20 |   pos ← pos + blocks[i]
21 return part
```

The implementation of $\Gamma S(x)$ (the combination of Algorithms 3.9 and 3.10) is displayed in Appendix B.3.

It can be demonstrated, by Theorem 3.1, that choosing the value for the parameter x according to the Lambert W function yields a reasonably high likelihood of obtaining a partition of a set that is close to or exactly the desired cardinality. By Theorem 3.1

we obtain as theoretical variance and standard deviation:

$$\begin{aligned}\mathbb{V}_x(N) &= x\frac{d}{dx}\mathbb{E}_x(N)\\ &= x\frac{d}{dx}(xe^x)\\ &= x(xe^x + e^x)\\ &= xe^x(x+1)\\ &= n(x+1)\end{aligned}$$

since $n = xe^x$. At the optimal value, $x = W(n)$, this gives

$$\mathbb{V}_x(N) = n(W(n)+1)$$

hence

$$\sigma = \sqrt{n(W(n)+1)}.$$

Using $n = 100$ as an example, we get that $\mathbb{V}_x(N) \approx 438.563014$ and consequently $\sigma \approx 20.941896$. It is known that for $n \in (0, \infty)$ we have $W(n) \sim \ln n$ and thus $\sigma \sim \sqrt{n \ln n}$ [1].

4 Comparitive Analysis

In the previous two sections, the algorithmic methods for random generation have been discussed in depth with several implementation examples. These examples were chosen to be the same between the two methods so that we may draw some comparisons between the strengths and shortcomings of each of the methods. As was mentioned, first and foremost we make note that the recursive method in general sees its largest shortcoming with regards to space requirements, which is not of concern to Boltzmann samplers. On the other hand, even for approximate size sampling,

Boltzmann samplers can require far more time to achieve the desired object when compared to the recursive method counterpart.

4.1 Binary Trees

The first most evident difference in performance with binary trees is the hit rate achieved, even in the pointed case, by the Boltzmann sampler. The standard deviation exceeds the expected size, and so we can infer that obtaining a tree, even with approximate size sampling, will likely require multiple runs. On the other hand, the amount of space required for the recursive method to produce a random tree of a given size becomes increasingly out of bounds for particularly large n, whereas in theory the Boltzmann sampler can produce such a tree without any issues insofar as space is concerned.

First we illustrate that the samplers achieve uniformity. One particular parameter that this can be measured by is the average depth of a tree of size n. The depth of a given node v_i in a tree is defined as the length of path which connects v_i to the root. By the depth of the tree itself we then mean the length of the maximum of these paths. The height of a tree is given by the depth + 1, the average height of a binary tree of size n is known to be asymptotic to $2\sqrt{\pi n}$, and hence the average depth is asymptotic to $2\sqrt{\pi n}$ [6]. Figure 4.1 compares the average depth of randomly generated trees among 1000 iterations using the recursive method sampler for each interger $n \in [1, 1000]$ (left) with the plot of $f(n) = 2\sqrt{\pi n}$ (right). We note some fluctuation for larger values of n, but generally similar plots are produced overall.

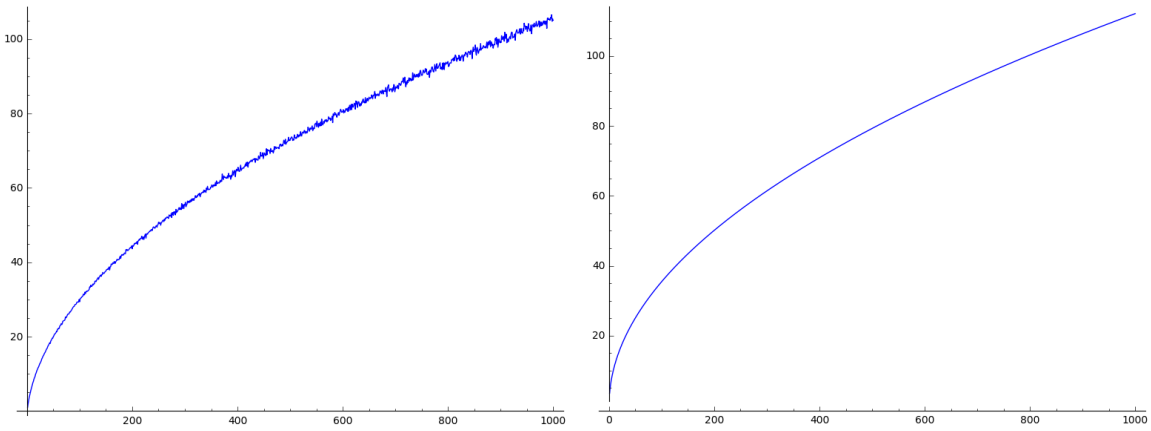

Figure 4.1: Average depths for binary trees on n internal nodes using the recursive sampler versus the asymptotic depth estimate.

.

Running the Boltzmann sampler 10000 times with the x parameter set accordingly for $n = 100$, 500, 1000, and 2000 yielded the resulting statistics: $\mu_{100} = 101.1883$, $\sigma_{100} = 139.8712$, $\mu_{500} = 500.2984$, $\sigma 500 = 707.4485$, $\mu_{1000} = 999.8445$, $\sigma_{1000} = 1419.1528$, $\mu_{2000} = 1993.1494$, and $\sigma_{2000} = 2807.1803$. The resulting historgrams can be seen in Figure 4.2. As expected, we note that the distributions are right skewed.

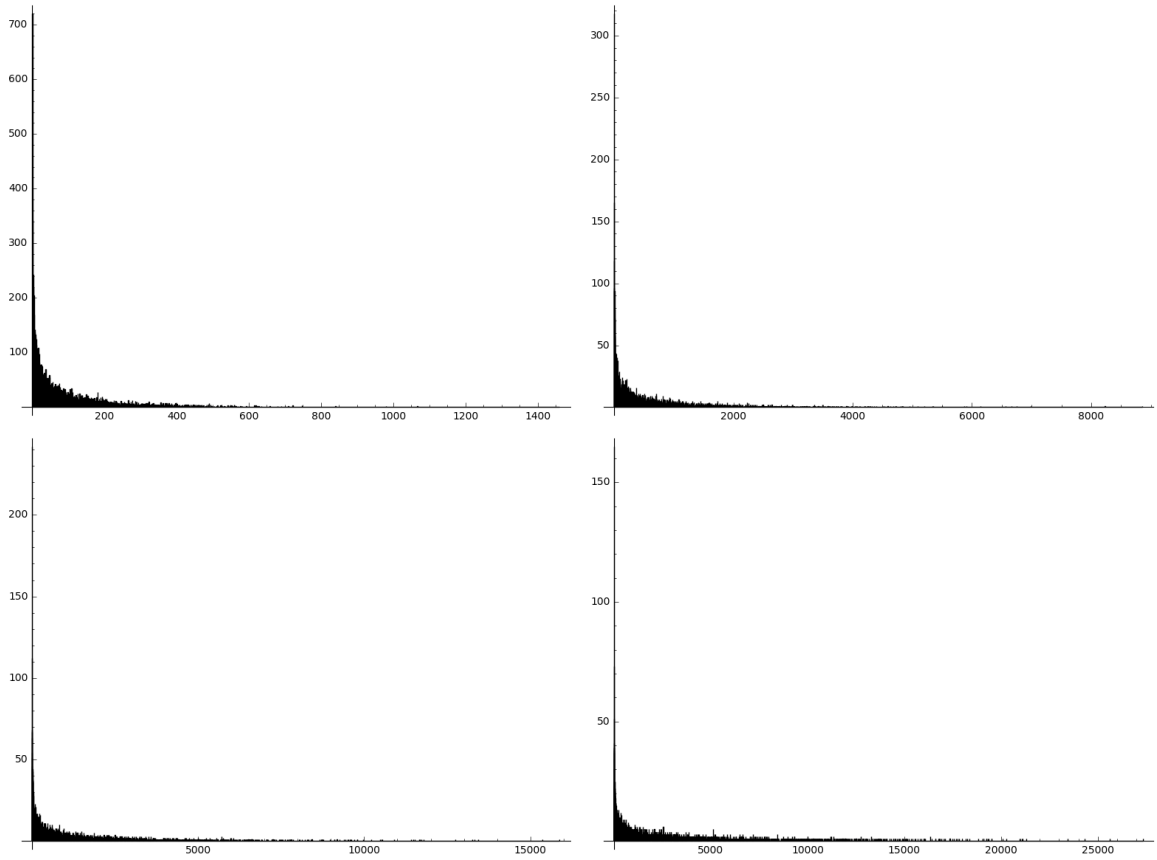

Figure 4.2: Histograms for 10000 iterations generating binary trees with Boltzmann sampler parameter set for $n = 100$, 500, 1000, and 2000.

We recall from Section 3.2 that the theoretical standard deviation for binary trees of size n using the Boltzmann sampler for pointed binary trees is $\sigma = \sqrt{2n^2 - 3n + 1}$. In Figure 4.3 we display the plot generated by calculating the standard deviation from 1000 iterations with the Boltzmann sampler for each integer $n \in [1, 500]$ (left) compared to the plot generated by $f(n) = \sqrt{2n^2 - 3n + 1}$ (right). We note that, although the plots do not match, it can be seen that the plot for the theoretical standard deviation is undoubtedly the line of best fit for our computed results.

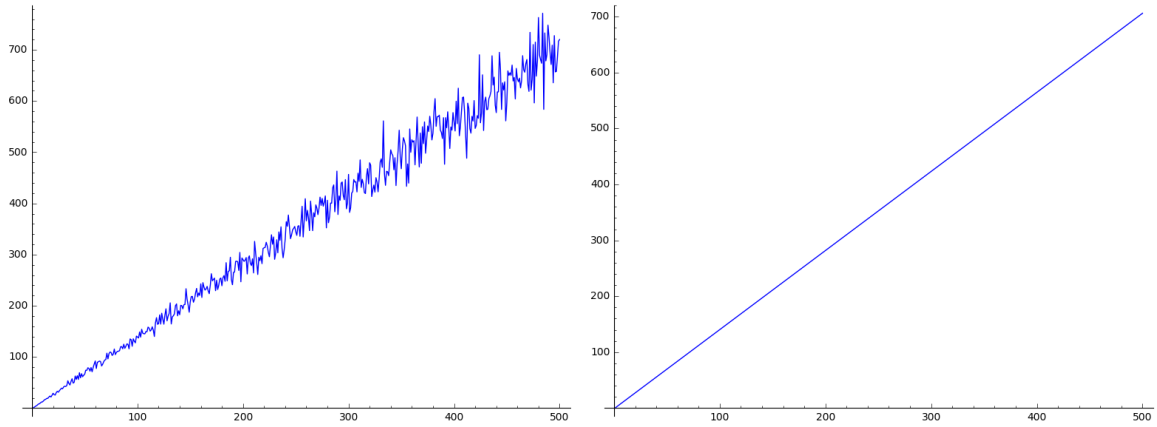

Figure 4.3: Standard deviation for trees of size n with Boltzmann sampling.

4.1.1 Timing

Timing is an import aspect to consider for samplers with respect to utility and efficiency. In Figure 4.4 we compare the average time (over 100 iterations) required to generate binary trees on n internal nodes for $n \in [1, 100]$ using the recursive method (left) and Boltzmann sampling (right). As expected, we see that the recursive method exhibits far superior performance with respect to time efficiency.

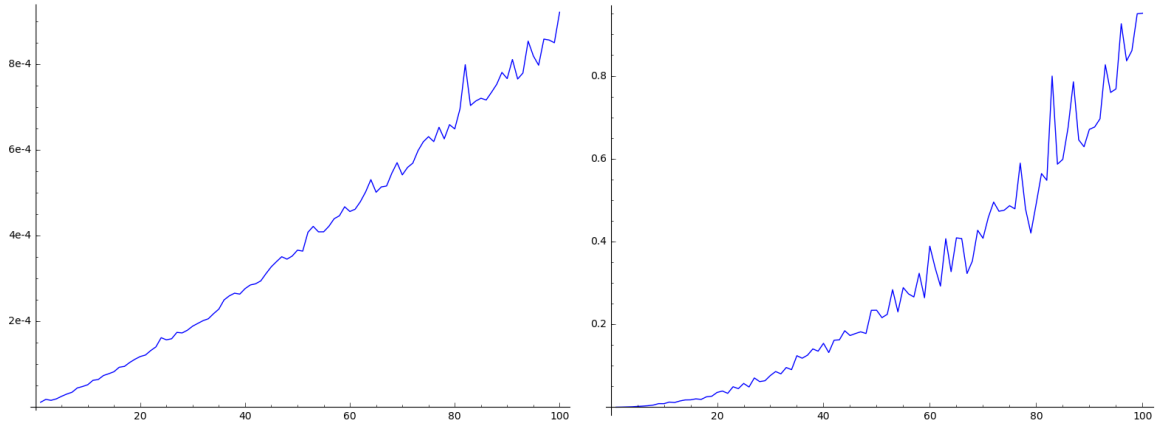

Figure 4.4: Average time required to generate a binary tree on n internal nodes: recursive method versus Boltzmann sampling.

Using approximate sampling, we can achieve better speed efficiency with Boltz-

mann sampling, albeit still inferior to the recursive method. Figure 4.5 demonstrates this with objects permitted to be within 5% and 10% of the desired size respectively. The same values of n and number of iterations were used as in the case of Figure 4.4.

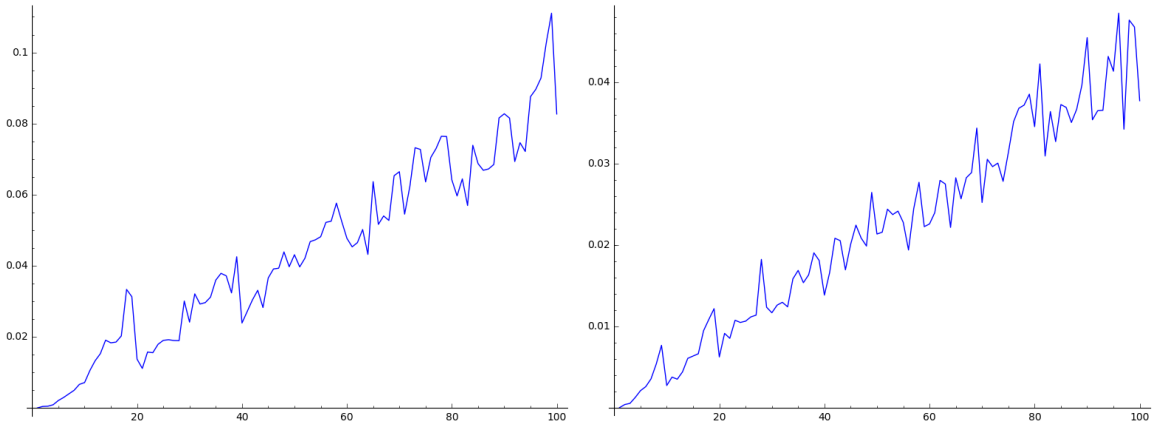

Figure 4.5: Time required to generate random binary trees on approximately n internal nodes.

4.2 Integer Partitions

In order to assess the uniform randomness of the integer partitions we generate, we can use the average number of parts in a partition of $n \in \mathbb{N}$ and compare this with the expected value. Kessler and Livingston demonstrated that the expected number of parts in a partition of n is asymptotic to

$$\frac{\sqrt{n}}{v} \ln n$$

where $v = \sqrt{\frac{2}{3}}\pi$ [8]. In Figure 4.6 we compare the results obtained from averaging 1000 interations using the recursive method sampler for partitions of each integer $n \in [1, 1000]$ with the plot given by $f(n) = \frac{\sqrt{n}}{v} \ln n$ and note the similarity between the plots.

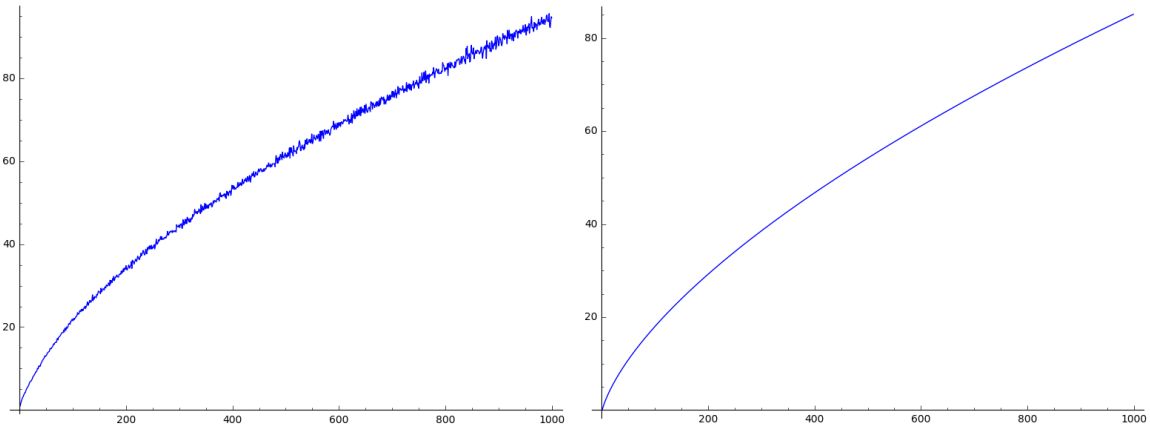

Figure 4.6: Average number of parts over 1000 iterations of the recursive method sampler versus expected number of parts.

Running the Boltzmann sampler 10000 times with the x parameter set accordingly for $n = 1000$ and 5000 yielded the resulting statistics: $\mu_{1000} = 990.3959$, $\sigma_{1000} = 223.3896$, $\mu_{5000} = 4974.6955$, and $\sigma_{5000} = 735.8441$. The resulting historgrams can be seen in Figure 4.7. As expected, the distributions appear relatively normal.

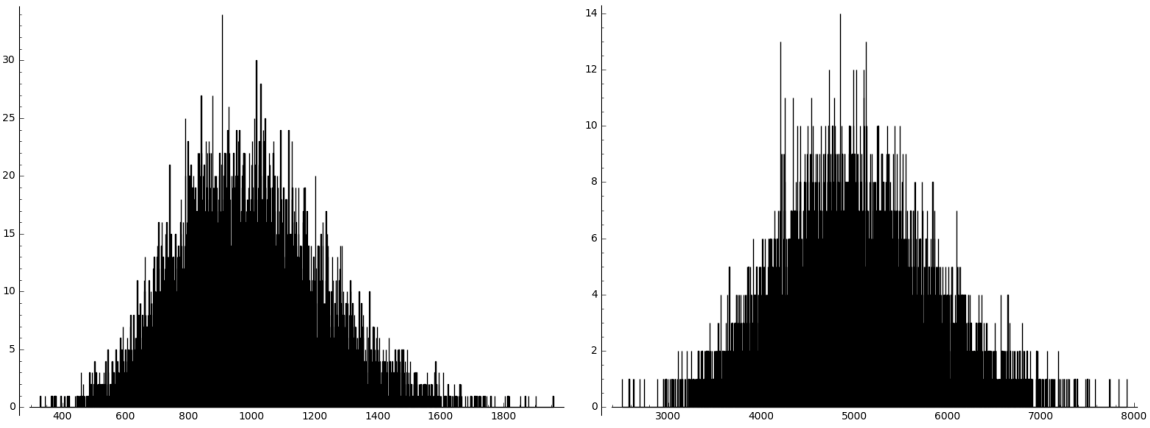

Figure 4.7: Histograms for 10000 iterations generating partitions of $n = 1000$ and 5000 with Boltzmann sampling.

We recall from Section 3.3 that our estimate for the theoretical standard deviation for integer partitions using Boltzmann sampling was $\sigma \approx \left(\frac{24}{\pi^2}\right)^{\frac{1}{4}} n^{\frac{3}{4}}$. In Figure 4.8 we display the plot generated by calculating the standard deviation from 1000 iterations

with the Boltzmann sampler for each integer $n \in [1, 500]$ (left) compared to the plot generated by $f(n) = \left(\frac{24}{\pi^2}\right)^{\frac{1}{4}} n^{\frac{3}{4}}$ (right). Though there does exist some variation (caused by fluctuations in the computed case), the curves displayed remain generally similar.

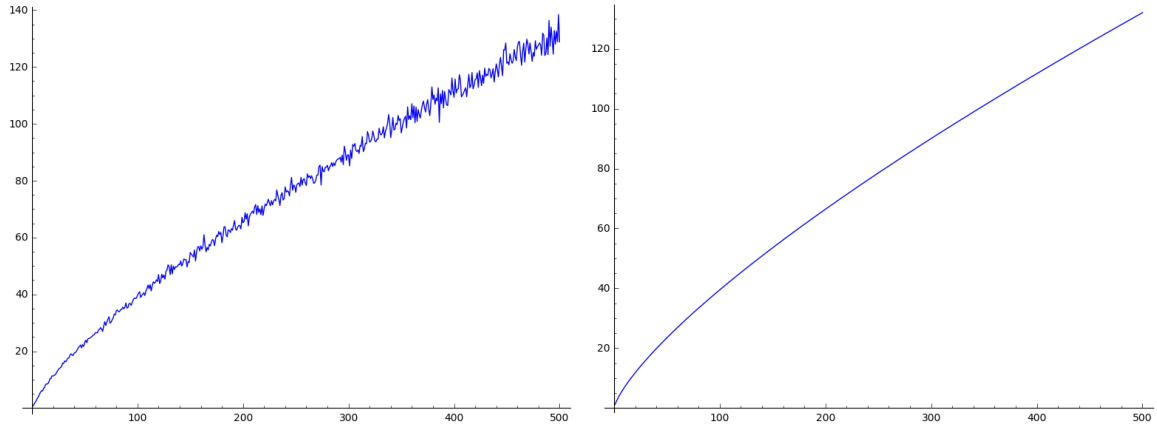

Figure 4.8: Standard deviation for partitions of n with Boltzmann sampling.

4.2.1 Timing

To assess timing performance, we compare the time required to generate uniformly random interger partitions using the recursive method and using Boltzmann sampling. In Figure 4.9 plots display the average time required to generate partitions of integers $n \in [1, 200]$. 100 iterations were used with the recursive method (left) and Boltzmann sampling (right). As expected we see clear superiority with respect to time efficiency from the recursive method.

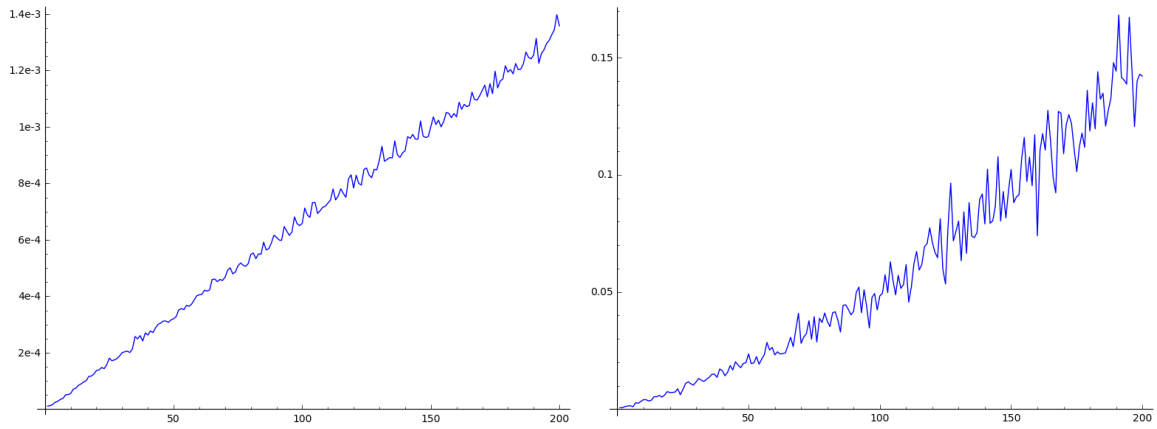

Figure 4.9: Time required to generate random partitions of n: recursive method versus Boltzmann sampling.

Using approximate sampling, we can achieve better speed efficiency with Boltzmann sampling. Figure 4.10 demonstrates this with objects permitted to be within 5% and 10% of the desired size respectively. The same values of n and number of iterations were used as in the case of Figure 4.9.

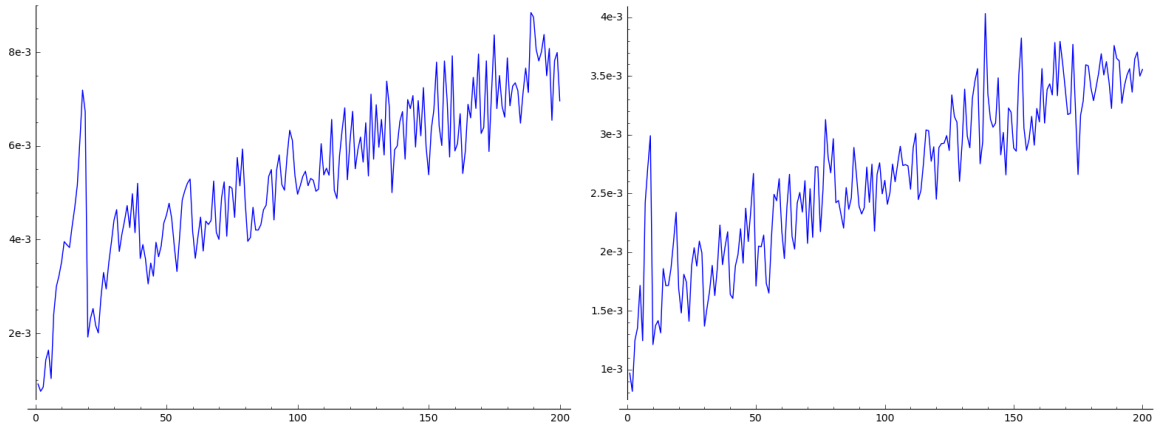

Figure 4.10: Time required to generate partitions of approximately n.

4.3 Set Partitions

To test for true uniform randomness in our generation of set partitions we can compare the average number of blocks over multiple iterations with a sampler with the actual

average number of blocks over all possibilities. The average number of blocks in a set partition of size n is given by $\frac{s_{n+1}}{s_n} - 1$. In Figure 4.11 we have the average number of blocks over 1000 iterations of the recursive method sampler for each integer $n \in [0, 300]$ (left) and the plot given by $f(n) = \frac{s_{n+1}}{s_n} - 1$ (right). We note that the two plots are nearly identical.

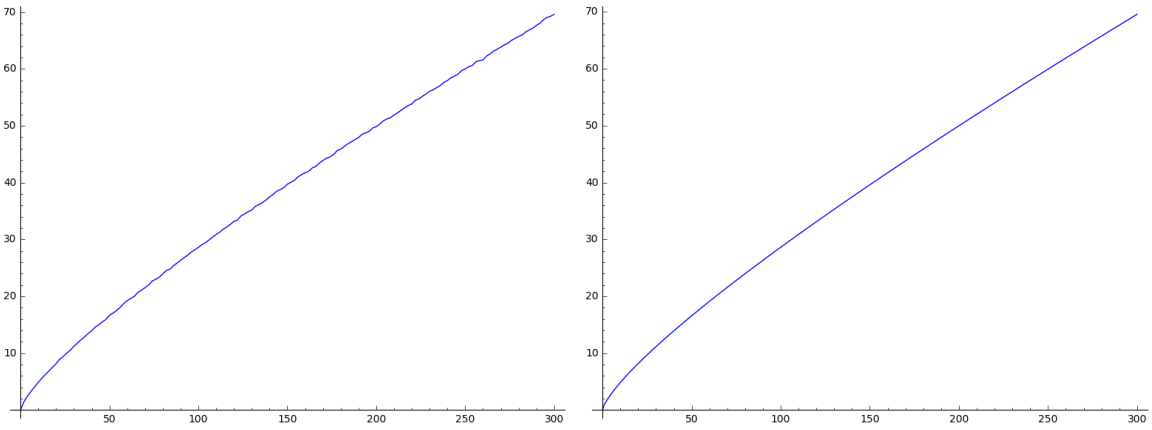

Figure 4.11: Average block sizes over 1000 iterations of the recursive method sampler versus actual average block sizes.

Running the Boltzmann sampler 10000 times with the x parameter set accordingly for $n = 100$ and 500 yielded the resulting statistics: $\mu_{100} = 99.8614$, $\sigma_{100} = 21.1103$, $\mu_{500} = 499.9603$, and $\sigma_{500} = 53.2078$. The resulting historgrams can be seen in Figure 4.12. Note that the distributions appear relatively normal, as expected.

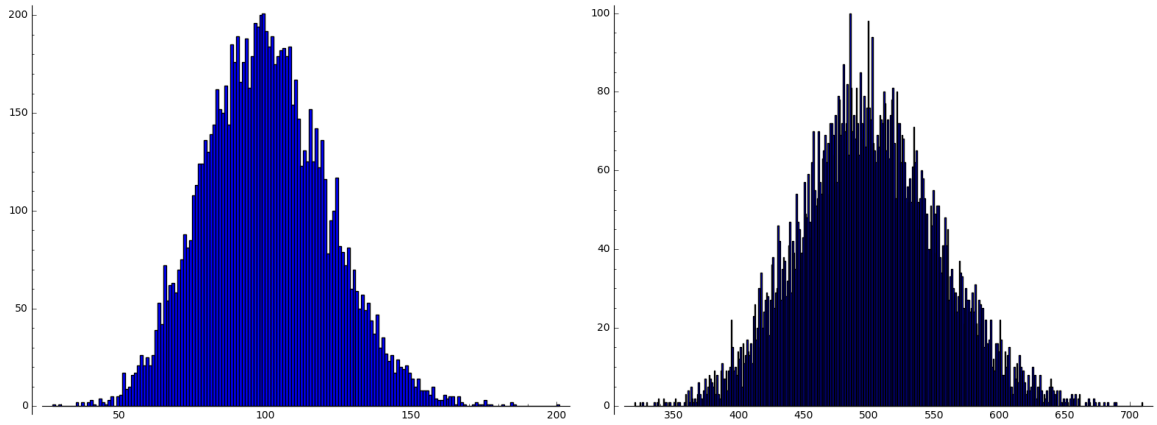

Figure 4.12: Histograms for 10000 iterations generating set partitions with Boltzmann sampler parameter set for $n = 100$ and 500.

We recall in Section 3.4 it was demonstrated that the theoretical standard deviation for set partitions using Boltzmann sampling was $\sigma = \sqrt{nW(n) + n}$. In Figure 4.13 we display the plot generated by calculating the standard deviation from 1000 iterations with the Boltzmann sampler for sets of cardinality n for each integer $n \in [1, 300]$ (left) compared to the plot generated by $f(n) = \sqrt{nW(n) + n}$ (right). Though there does exist some variation (caused by fluctuations in the computed case), the curves displayed remain generally similar.

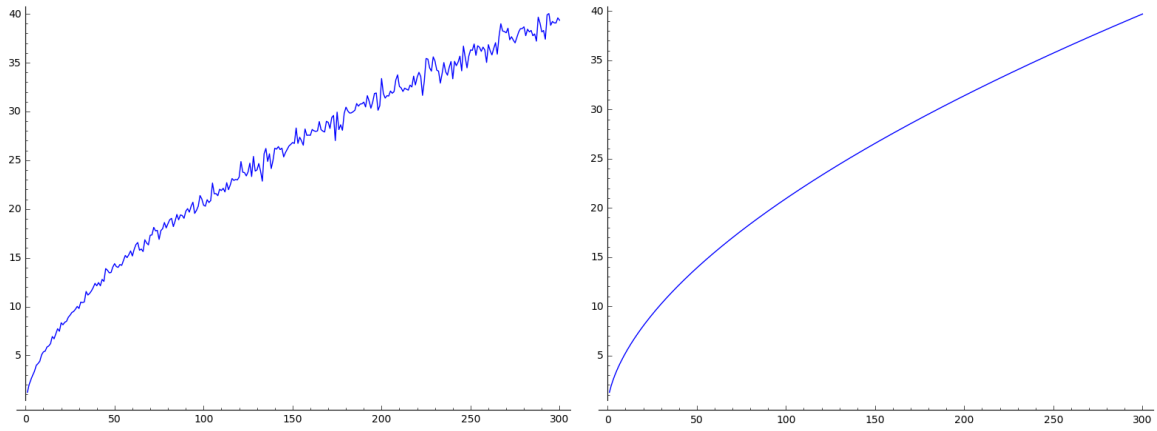

Figure 4.13: Standard deviation for sets of cardinality n with Boltzmann sampling.

4.3.1 Timing

To assess timing performance, we compare the time required to generate uniformly random set partitions using the recursive method and using Boltzmann sampling. In Figure 4.14 plots display the average time required to generate partitions of sets of size n for $n \in [1, 100] \cap \mathbb{Z}$. 100 iterations were used with the recursive method (left) and Boltzmann sampling (right). We see, as expected, that the recursive method exhibits superior speed performance on average.

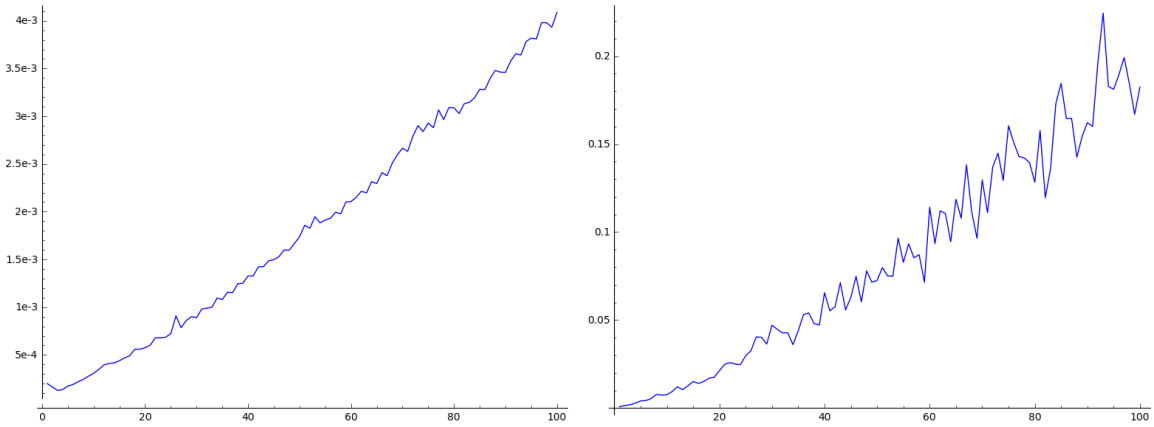

Figure 4.14: Time required to generate random sets of cardinality n: recursive method versus Boltzmann sampling.

Using approximate sampling, we can achieve better speed efficiency with Boltzmann sampling. Figure 4.15 demonstrates this with objects permitted to be within 5% and 10% of the desired size respectively. The same values of n and number of iterations were used as in the case of Figure 4.14.

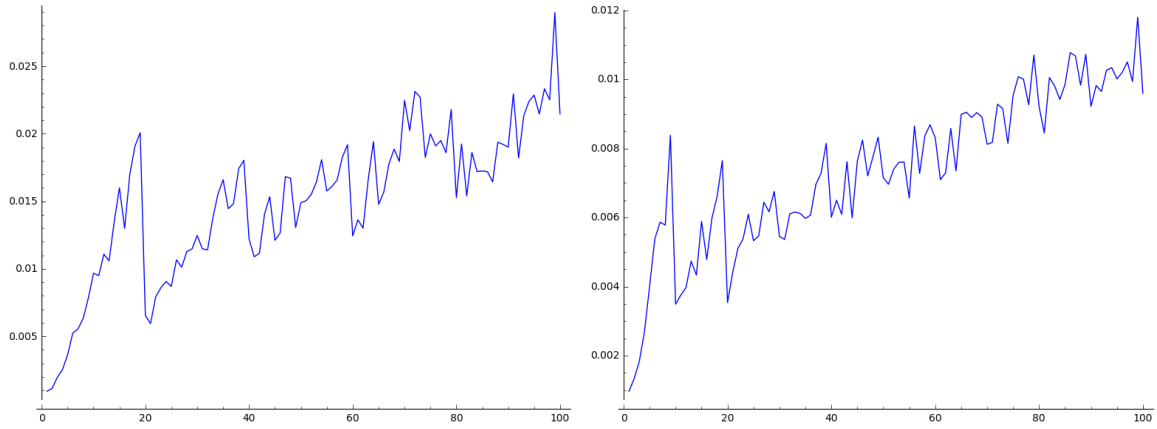

Figure 4.15: Time required to generate random sets of approximately cardinality n.

4.4 General Timing Comparison

As discussed and demonstrated in Sections 4.1.1, 4.2.1, and 4.3.1, the recursive method exhibits superiority to the Boltzmann method when concerning time efficiency in sampling from $\mathscr{C}_n$. Indeed sampling from $\mathscr{C}_n$, the recursive method sampler will generate an object in $O(n \log n)$ time with $O(n^2)$ or $O(n^{1+\varepsilon})$ $(0 < \varepsilon < 2)$ integer arithmetic operations (pre-processing time) required for the array of values from the counting sequence to be computed [9, 4].

On the other hand, to sample from $\mathscr{C}_n$ using the Boltzmann model, we require $O(n^2)$ time per generation in the worst case scenario. Using strategies introduced by Flajolet et al., this can be reduced to $O(n^{1+\varepsilon})$ $(0 < \varepsilon < 2)$ or even $O(n)$ time per generation for certain classes. If instead we want to sample from $\mathscr{C}_N$ (approximate size sampling) then we can achieve $O(n)$ time per generation using the Boltzmann method given some interval $[n(1-\varepsilon), n(1+\varepsilon)]$ for a particular $\varepsilon > 0$. With this in mind however, the optimal scenarios for Boltzmann sampling occur under the assumption that we have an oracle that can perform efficient and perfectly accurate floating point arithmetic. Thus we require floating point approximations in practice which results in some small amount of preproccessing time for both the exact and approximate size

sampling cases. Typically this time is $O((\log n)^k)$, where k is small [4].

4.5 Space

As discussed in Sections 1 and 2, the recursive method's primary shortcoming lies within the requirement to compute and store the values $\{c_k\}_{k=0}^{n}$. This requires $O(n)$ large integers to be stored in memory. Boltzmann samplers on the other hand require $O(1)$ constants to be stored in memory, regardless of the desired object size [4]. Therefore, the only space constraint for Boltzmann samplers is that there be enough memory to store the object itself. As a result there is a value of n for which the recursive method will be unable to sample from $\mathscr{C}_n$ due to space constraints, where the Boltzmann method will, in theory, remain capable. Of course we may reach a point where, due to time constraints, only approximate size sampling may be feasible.

5 Conclusions

It is evident that both the recursive method and the Boltzmann method for uniformly random generation of combinatorial structures possess strengths and weaknesses as well as applications. It is clear that, when time constraints are of concern, the recursive method reigns superior. However if we are in need of generating especially large objects, it may only be possible via Boltzmann sampling. For particularly large objects, approximate sampling becomes more useful as well, thus again promoting the use of Boltzmann samplers. Applications and future work shall now be addressed.

5.1 Applications

It is important to note the applications of the end results of implementing these algorithmic methods. Of course one can state that these methods allow us to know what various structures of a given size look like on average, but it is important to see

how this applies to a variety of other problems.

5.1.1 Binary Trees

Binary trees are used to model various data structures as well as searching and sorting algorithms in computing science. Random binary search trees and the related objects, *treaps*, are used in developing efficient binary searching algorithms. Furthermore, the famous *Quicksort* algorithm is related to and modelled by random binary trees [5].

In theory, we can expand on the samplers used for binary trees to develop samplers for more general trees. Randomly generated trees have been applied to a variety of problems. For example, caching protocols that mitigate the issue of *hot spots* on the Internet as well as search algorithms [7].

5.1.2 Integer Partitions

Random integer partitions can be applied to various areas of mathematics to gain insight. One such area is with regards to the general linear and the symmetric group, denoted $S(n)$. Integer partitions index irreducible representations of $S(n)$. Moreover, the *Plancherel measure* on the set of partitions of n is a probability measure that is closely related to the Fourier transform of $S(n)$ [10].

The number of ways in which we can grow the Young diagram of a partition λ from the empty partition to n while maintaining a partition has numerous applications to understanding various growth processes in representation theory among other fields [10]. It is evident that understanding the average way we expect to grow a partition can be discovered through uniform sampling of partitions.

5.1.3 Set Partitions

Set partitions are equivalent to equivalence relations on a set. Therefore, sampling uniformly from all partitions of a set of cardinality n lends information regarding the

equivalence relations on that same set [11].

5.2 Future Research

First and foremost it is evident that making use of the applications discussed in Section 5.1 would be of the utmost importance for future research. Both the example classes considered, as well as a multitude of other combinatorial classes, have countless applications to a variety of problems. Understanding general properties of these classes is possible through uniform sampling.

It must be noted that there exists a package in Maple called CombStruct which implements the recursive method, allowing for user specified classes so long as they meet the decomposition requirements. It then stands to reason that another important area of future work would be to develop such a package or library that would implement the Boltzmann sampling method in a similar fashion. The major difference in this respect would be the requirement of the closed form expression for the classes generating function, as well as a general definition of the structure of objects belonging to the class considered.

6 References

[1] R. M. Corless, G. H. Gonnet, D. E. Hare, D. J. Jeffrey, and D. E. Knuth. On the Lambert W function. *Advances in Computational mathematics*, 5(1):329–359, 1996.

[2] S. DeSalvo. Improvements to exact Boltzmann sampling using probabilistic divide-and-conquer and the recursive method. *Pure Mathematics and Applications*, 26(1):22 – 45, 2017.

[3] P. Duchon. Random generation of combinatorial structures: Boltzmann samplers and beyond. In *Proceedings of the 2011 Winter Simulation Conference (WSC)*, pages 120–132, Dec 2011.

[4] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Boltzmann samplers for the Random generation of Combinatorial Structures. *Combinatorics, Probability and Computing*, 13(4-5):577–625, 2004.

[5] P. Flajolet, X. Gourdon, and C. Martínez. Patterns in random binary search trees. *Random Structures & Algorithms*, 11(3):223–244, 1997.

[6] P. Flajolet and A. Odlyzko. The average height of binary trees and other simple trees. *Journal of Computer and System Sciences*, 25(2):171 – 213, 1982.

[7] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, volume 97, pages 654–663, 1997.

[8] I. Kessler and M. Livingston. The expected number of parts in a partition of n. *Monatshefte für Mathematik*, 81(3):203 – 212, Sep 1976.

[9] A. Nijenhuis and H. S. Wilf. *Combinatorial algorithms for computers and calculators*. 1978.

[10] A. Okounkov. The uses of random partitions. In *XIVth International Congress on Mathematical Physics*, pages 379–403. World Scientific, 2006.

[11] J. Pitman. Some probabilistic aspects of set partitions. *The American mathematical monthly*, 104(3):201–209, 1997.

[12] P. Stein and M. Waterman. On some new sequences generalizing the Catalan and Motzkin numbers. *Discrete Mathematics*, 26(3):261 – 272, 1979.

Appendices

Appendices A - B contain the source code for the sampler examples considered in this thesis. All code is written in the python-based mathematical system SAGE math. In order to attain particularly large objects using the recursive method, it is in some cases necessary to import the sys module and manually change the python recursion limit. This also applies to the Boltzmann sampler for binary trees, due to the inherently recursive nature of the class.

A Recursive Method Source Code

A.1 Binary Trees

```
def genBinaryTree(n):
    if n == 0:
        return
    elif n == 1:
        return BinaryTree([])
    else:
        x = randint(1, b[n])
        tally = 0
        for i in range(0, n):
            tally += b[i] * b[n-i-1]
            if x <= tally:
                k = i
                break
        return BinaryTree([genBinaryTree(k),genBinaryTree(n - k - 1)])
```

A.2 Integer Partitions

```
import random

def makeTable(n):
    p = [[0 for x in range (n+1)] for y in range(n+1)]
    for m in range (0, n+1):
        for k in range (0, n+1):
            if m == 0:
                p[m][k] = 1
            elif k == 0:
                p[m][k] = 0
            else:
                p[m][k] = p[m-k][k] + p[m][k-1]
    return p
```

```
def genPart(n, k, p, part):
    if n < 1:
        return part
    u = random.uniform(0,1)
    x = p[n-k][k] / p[n][k]
    if u < x:
        part.append(k)
        return genPart(n - k, k, p, part)
    else:
        return genPart(n, k - 1, p, part)
```

A.3 Set Partitions

```
import random

def Rand_Permutation(arr):
    n = len(arr)
    for i in range(0,n-1):
        j = random.randint(i,n-1)
        temp = arr[i]
        arr[i] = arr[j]
        arr[j] = temp

def chooseBlocks(m, bn, b):
    if m <= 0:
        return b

    x = randint(1, bn[m])
    k = 1
    tally = 0
    for i in range (0, m):
        tally += binomial(m-1, k-1) * bn[m-k]
        if x <= tally:
            break
        k = k + 1
    b.append(k)
    return chooseBlocks(m - k, bn, b)

def RandSetPart(n):
    if n == 0:
        return []
    s = range(1,n+1)
    bn = []
    for i in range (0, n+1):
```

```
        bn.append(bell_number(i))

    blocks = chooseBlocks(n, bn, [])
    Rand_Permutation(s)
    part = []

    pos = 0
    for i in range(0,len(blocks)):
        temp = []
        for j in range(0,blocks[i]):
            if i == 0:
                temp.append(s[j])
            else:
                temp.append(s[j+pos])
        part.append(temp)
        pos = pos + blocks[i]

    return part
```

B Boltzmann Samplers Source Code

B.1 Binary Trees

```
import random

def GammaB(x):
    global nodes
    p0 = (2*x) / float(1 - math.sqrt(1 - 4*x))
    u = random.uniform(0,1)
    if u < p0:
        nodes += 1
        return
    else:
        return BinaryTree([GammaB(x),GammaB(x)])

def GammaBP(x):
    global nodes
    p1 = math.sqrt(1 - 4*x)
    p2 = 0.5 - 0.5*(math.sqrt(1 - 4*x))
    u = random.uniform(0,1)
    if u < p1:
        nodes += 1
        return
    elif p1 <= u < p1 + p2:
        return BinaryTree([GammaBP(x), GammaB(x)])
```

```
    else:
        return BinaryTree([GammaB(x), GammaBP(x)])
```

B.2 Integer Partitions

```
import random

def RandIntPart(n):
    if n == 0:
        return []
    x = float(e**(-pi/sqrt(6*n)))
    part = []
    p = 1
    for i in range(1, n+1):
        p = p*x
        while True:
            u = random.uniform(0,1)
            if u < p:
                part.append(i)
            else:
                break
    return part
```

B.3 Set Partitions

```
import random

def Poisson(x):
    u = random.uniform(0,1)
    k = 0
    tally = 0
    while True:
        if u < tally:
            return k - 1
        else:
            tally += e**(-x) * (x**k/float(factorial(k)))
            k += 1

def Poisson_NonEmpty(x):
    u = random.uniform(0,1)
    k = 1
    tally = 0
    while True:
        if u < tally:
            return k - 1
        else:
```

```
            tally += (1/float(e**x - 1)) * (x**k/float(factorial(k)))
            k += 1

def Rand_Permutation(arr):
    n = len(arr)
    for i in range(0,n-1):
        j = random.randint(i,n-1)
        temp = arr[i]
        arr[i] = arr[j]
        arr[j] = temp

def RandSetPart(m):
    x = lambert_w(float(m))
    k = Poisson(e**x - 1)
    blocks = []
    n = 0
    for i in range(0, k):
        sel = Poisson_NonEmpty(x)
        blocks.append(sel)
        n += sel

    a = range(1,n+1)
    Rand_Permutation(a)
    part = []

    pos = 0
    for i in range(0,len(blocks)):
        temp = []
        for j in range(0,blocks[i]):
            if i == 0:
                temp.append(a[j])
            else:
                temp.append(a[j+pos])
        part.append(temp)
        pos = pos + blocks[i]
    return part
```