University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Final-Year Project Report

BSc in Computer Science

Python Package for Enumerating Combinatorial Objects

Donnchadh Durkan

Department of Computer Science

Supervised by

Dr Kieran T. Herley

Department of Computer Science

April 2020

Abstract

The goal of the project is to create a programmer-friendly Python package that allows developers to access useful combinatorial algorithms. The package aims to build on the foundations of Donald Knuth's work in his book The Art of Computer Programming Volume 4A: Combinatorial Algorithms, and provides efficient implementations of several of the algorithms discussed there.

Specifically, the package will include the ability to generate all permutations and combinations of a sequence, generate all trees for a given number of vertices, generate all partitions for a given integer, and all partitions for a given set. In addition to this functionality, the package will also allow for the random generation of a mapping of the objects mentioned. The algorithms will be tested against implementations provided by existing Python packages, such as itertools, which will provide a useful benchmark for their efficiencies, and give expectations for how the algorithms should function in terms of inputs and outputs.

Keywords: Combinatorics, permutations, combinations, partitions, trees.

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: Donnchadh Durkan Donnchadh Durkan

Date: Wednesday 6th April, 2022

Acknowledgements

I wish to express sincerest thanks to Dr Kieran T. Herley for his valuable and constructive feedback throughout the project. His unwavering dedication to provide assistance has been greatly appreciated.

I would also like to thank David Fox and Tim Lehane for their continued friendship. They have been a tremendous source of positivity and happiness.

Finally, I wish to thank my parents, as well as my partner, for their support and encouragement.

Contents

1 Introduction

1.1 Overview of Combinatorics

Combinatorics describe the different patterns a finite set or structure can be arranged into. These patterns are often referred to as combinatorial objects. A combinatorial object is an object which has a one-to-one, or injective, mapping to a finite set of integers. Some examples include permutations, combinations and partitions of sequences.

The ability to generate these objects is useful in a variety of areas. Algorithm design and analysis relies on the ability to generate test data from combinatorial objects. This allows the calculation of average and worst-case complexity, which can drive the improvement of the algorithms. Alternatively, some problem spaces are too large to rely on deterministic algorithms, as they lead to unrealistic run-times. A solution to this is to use randomized or approximation algorithms, which aim to reduce the problem space. These algorithms lean heavily on combinatorial methods for analysis.

1.2 Project Aims

The goal of the project is to create a programmer-friendly Python package that allows developers to access useful combinatorial algorithms. The package primarily aims to build on the foundations of Donald Knuth's work in his book *The Art of Computer Programming Volume 4A: Combinatorial Algorithms*, and provides efficient implementations of several of the algorithms discussed there. Specifically, the package will include the ability to generate all permutations and combinations of a sequence, generate all trees for a given number of vertices, generate all partitions for a given integer, and all partitions for a given set. In addition to this functionality, the package will also allow for the random generation of a mapping of the objects mentioned.

Itertools is an existing Python package which provides permutation and combination algorithms that mirror certain aspects of the functionality required [**?**]. As a result, *itertools* will serve as a useful benchmark for the new package. All new algorithms should be tested against it, and should provide at least the same functionality. This package aims to extend beyond *itertools'* capabilities, and should encourage linked functionality between the different combinatorial areas of interest.

1.3 Deliverables

The project was achieved by approaching the task one combinatorial object at a time. In the case of permutations and combinations, Knuth provided numerous algorithms which all served the same purpose. Therefore careful study of his writings and analysis was needed in order to reduce the number of algorithms as potential candidates for the package.

Once these algorithms were selected, they were implemented in Python and rigorously test and compared against each other. It is rare that one algorithm outperforms all others in every scenario, therefore it was imperative that the algorithms were considered on several different merits.

The end result of the project is a package which proudly offers effective and efficient implementations of Knuth's algorithms, as outlined previously. Despite the utilization of C in *itertools*, which will be discussed later, this package boasts competitive algorithms for generating permutations and combinations.

2 Analysis

2.1 Combinatorial Objects

To fully understand the aims and capabilities of the package, it is important to have an insight into different combinatorial objects. There are four main objects covered by Knuth that the package aims to have implementations for.

2.1.1 Permutations

A permutation is an arrangement of a finite number of items in a set. Subsequently the order that the items appear in is the defining aspect of a permutation. Permutations are generated by altering this order. Take the set {A,B,C}, where $n = 3$. All possible permutations of size $k = 3$ of this set are:
(A,B,C), (A,C,B), (B,A,C), (B,C,A), (C,A,B), (C,B,A)

This package predominantly deals with the case where $n = k$. Consequently, the calculation for the number of permutations of size n is trivial. As the order matters, there are n options for the first item, $n-1$ for the second item and so on, down until there is only one option for the final item. Putting this together gives:

$$n(n-1)(n-2)...(1) = n!$$

However, when $n \neq k$, the factorial process must be stopped after multiplying $n-k$ terms. Mathematically, the simplest way to achieve this is to divide by $(n-k)!$, leaving the formula:

$$\frac{n!}{(n-k)!}$$

Factorial functions grow at a faster rate than exponential functions, with the exception of the superexponential n^n. While the example above only has six permutations, increasing n from $n = 3$ to $n = 10$, increases the number of permutations to 3,628,800.

This rapid growth ensures that any algorithms in this package must provide intuitive ways to interact with the permutations.

2.1.2 Combinations

Combinations are not dissimilar to permutations, with the exception being that order of the elements in the set does not matter. Often, there is a desire to know how many subsets of size k can be taken from a set of size n.

Again, take the set A,B,C with $n = 3$ and $k = 3$. Therefore, there is only 1 possible combination:
(A,B,C)

However, for the same set, but $k = 2$, we get 3 combinations:
(A,B), (A,C), (B,C)

The number of combinations is calculated building off of the formula for the number of permutations, which works out as:

$$\frac{n!}{k!(n-k)!}$$

2.1.3 Partitions

There are two type of partitions, integer partitions and set partitions. In the case of the former, the process is characterized taking a positive integer n, and breaking it up in the sum of positive integers. For $n = 3$, the partitions are:
(3), $(2 + 1)$, $(1 + 1 + 1)$

A major use case for integer partitions are algorithms that calculate the amount of change required to give a customer upon payment. By constraining the integers that can appear in the partition to represent monetary values, a greedy approach can be taken to arrive quickly at the optimal solution, that is, the least amount of notes and coins required.

Set partitions refer to the division of a set into subsets. A partition of a set assigns each element in the set to exactly one subset. The subsets are non-empty, they must contain at least 1 element. The amount of subsets that exist is based on the number of partitions. 1 partition means 2 subsets, 2 partitions means 3 subsets, and so on. It is crucial to note that subsets are not necessarily proper subsets, {{1,2,3}} is a valid partition of {1,2,3}.

2.1.4 Trees & Forests

Trees and forests are similar structures, they are vertex-edge graphs. Any vertex can have at most one parent, but can have multiple children. The primary difference between the two structures is that there is exactly one path between any 2 vertices in a tree, however there may not be a path between any two vertices in a forest. In this context, forests are acyclic, and both trees and forests are undirected.

Structures like these have many uses, such as storing hierarchical data. They are dynamic, meaning it is simple to add and remove vertices from them as needed. Due to their non-linear shape, it is harder to generate all trees or forests. This is reflected in the design and implementation of the algorithms, which will be discussed later on.

2.2 Shaping the Package

In order to decide the specific functionality of the package, careful analysis of existing material as well as *itertools* needed to be completed. The investigation began by looking at *The Art of Computer Programming Volume 4A: Combinatorial Algorithms.* Permutations were explored first, and it became apparent that an adequate algorithm for generating uniformly random permutations was not mentioned in the book. Further research in this area led to an algorithm popularized by Knuth, known as *Knuth Shuffles* [?], but in spite of its name, it was originally designed in 1938 by Fisher & Yates [?].

Combinations is the only area of the project where the package does not match up to *itertools* capabilities. Knuth usually examines combination generation algorithms where replacement of objects in the set does no occur. Subsequently this package does not provide the functionality for combination with replacement.

Much like permutations, Knuth does not write about algorithms for generating uniformly random combinations. Fortunately, Knuth does propose such an algorithm in *The Art of Computer Programming Volume 2, Seminumerical Algorithms* [?].

Itertools only provides set partitioning through an unofficial extended toolset [?]. Adding Knuth's work to the package is an extension beyond what is available in *itertools.* Knuth provides algorithms for integer partitioning, set partitioning and multiset partitioning. A random set partition algorithm by Nijenhuis & Herbert [?] completes the partition algorithms. With regards to tree generating algorithms, Knuth fully covers all aspects of this in the book, and as such there is no need for further research into this area.

Knuth does not scrutinize these algorithms' complexities. By implementing several algorithms for the tasks, it becomes easier to evaluate them against each other.

2.3 Python Implementation

It is important to note that *itertools* is a Python API for a C backend. Therefore, it is difficult to meaningfully compare *itertools'* algorithms run times to a pure Python implementation, due to Python being an interpreted language as opposed to C which is compiled. *Cython*, a static compiler for Python [?], should be utilised in order to speed up run times by essentially stripping back some Pythonic features such as dynamic typing. The use of *Cython* will not guarantee matching times with *itertools*, but the intention is that it will provide competitive times.

While *Cython* solves some issues, it creates others. *Cython* does not support generator functions as part of `cdef` functions. Immediately this limits the benefits that are achievable using *Cython* if generators are required. Alternatively, rather than using generators, it may be possible to use *numpy* arrays to store sequences. This would give enormous gains in runtime, but despite *numpy* using less memory, that concern remains.

As explained in section 2.1.1, the number of objects being generated can quickly reach cumbersome levels. It is in feasible to create arrays containing millions of objects, and store them in memory. To counteract this issue, the package makes use of Python's *generator* functions [?]. They are functions which return generator objects, allowing developers to generate the next item in a sequence only when called to do so.

The package attempts to maintain uniform usability throughout. Input sequences for the functions can take the form of any iterable object. This means that the generator object returned from any of the functions in the package can be used as an input for several of the other functions, achieving the linked functionality.

3 Design

3.1 Technologies

For the most part, the package does not depend on any libraries outside of the default Python libraries. The exception to this being the *random* library, as a source of entropy is required to generate uniformly random combinatorial objects.

As discussed previously, permutation and combination algorithms employ the use of the static compiler *Cython* to improve the time taken to generate the next object in the sequence. The package aims to be as lightweight as possible, by avoiding all unnecessary overheads.

3.2 Testing Generator Algorithms

The initial method for testing the algorithms begins by simply reading Knuth's work. He often discusses algorithms which are not necessarily the most efficient, simply because they take a unique approach. Following this idea, it is possible to rule out several algorithms for this reason, or due to unnecessary requirements to use them. Algorithm G for generating permutations [?] is a good example of this, where a Sims table is required for the algorithm to work. This needless overhead is a hindrance to a developer in nearly all cases.

Once the remaining algorithms are chosen and implemented, it follows logically that they should be timed, given Knuth's evasion of complexities. The next question is how should the algorithms be timed, where there are two possible approaches.

1. Use the generator functions to step through the algorithm one object at a time. Record the time it takes to generate the next object.

2. Wrap the function call in Python's built in function `list()`, which will automatically create a list containing all the objects that the generator will output. The total time taken from start to finish is recorded, and this is divided by the number of items in the list to get the mean time taken per object generation.

Option two appears to be the better approach. The first option will lead to a large number of data points, which can be difficult to maintain when the input sequence is large. Furthermore, it is also possible that option is affected by outliers due to patterns in the sequences that may occur later on. As a result, option two will give a more accurate estimate of the mean time taken per object generated.

This value can then be compared against the values from the other algorithms, as well as those provided by *itertools*, and the *Cython*-refactored algorithms to determine which will return the best average use case.

3.3 Testing Random Algorithms

Despite formal proofs of randomness being given for the algorithms used in this package, there is no guarantee that the Python 'translations' maintain this property. Testing the randomness of algorithms can be challenging given that their outputs are inherently unpredictable, so unit testing is not the best method.

One method involves running these algorithms with a relatively small input sequence, for example a sequence of size four. The algorithms would run repeatedly for 100,000 iterations, where the frequency of the results would be tracked. An input sequence of size four is advantageous compared to a greater size, as it is more likely to lead to a better distribution of values, and there number of possible values is more manageable ($4! = 24$ possible permutations in this case). 100,000 iterations is chosen to ensure a good distribution of values also. It will also serve to highlight any significant anomalies in the actual frequencies. By graphing the actual frequencies against the expected frequencies, any major discrepancies would be clearly visible on graph. This would act like an informal examination of the results to determine randomness.

While the visual aid of the graph is useful, it is too vague to be able to draw any real conclusions. The chi-squared test is a hypothesis test which evaluates the relationship between expected and actual frequencies of a given category. This makes it an excellent candidate for assessing the Python algorithms randomness. By interpreting the $p-value$ returned by the test, a clearer idea is given about the true randomness. Although this is not a formal proof, it should serve satisfactorily to verify the integrity of the algorithms.

4 Implementation

4.1 Software Implementations

The structure of the algorithms remain simple at their core. They were implemented using as many of Python's native features as possible. This is because Python offers excellent built-in methods for objects like lists, which feature widely throughout the package. Efficient manipulation of lists is key in achieving optimal results.

Please note that some algorithms may have the same name, due to their appearance in different sections of the book.

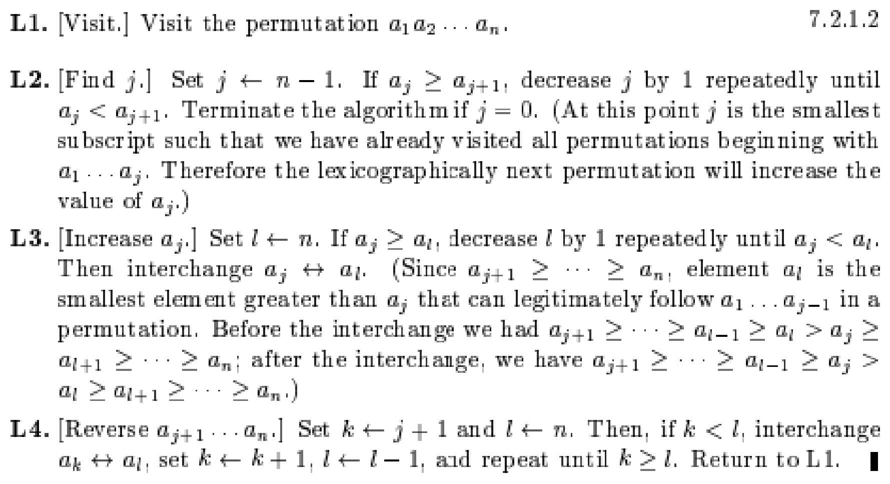
4.1.1 Permutations

Four algorithms for generating permutations were chosen to be evaluated.

1. The first algorithm [?], aptly named Algorithm L, visits the permutations in lexicographic order. Lexicographic order is advantageous as it essentially represents counting in a variable base. It is familiar to the user and is often one of the only approaches where you can predict the next sequence that will be generated.

   This algorithm has an interesting property, where the input sequence must be sorted. This is not a desirable feature, as sorting can be time-consuming and it can not deal with sequences containing duplicates.

   To work around this issue, the algorithm creates a new array `[0, 1, ..., n-1]` where n is the length of the input sequence. The algorithm then alters this array, rather than the original. By taking this approach, a sorted sequence is guaranteed containing no duplicated. This array acts as indices for the original. Using Python's list comprehension, the generator object can serve



L1. [Visit.] Visit the permutation $a_1 a_2 \ldots a_n$. 7.2.1.2

L2. [Find j.] Set $j \leftarrow n - 1$. If $a_j \geq a_{j+1}$, decrease j by 1 repeatedly until $a_j < a_{j+1}$. Terminate the algorithm if $j = 0$. (At this point j is the smallest subscript such that we have already visited all permutations beginning with $a_1 \ldots a_j$. Therefore the lexicographically next permutation will increase the value of a_j.)

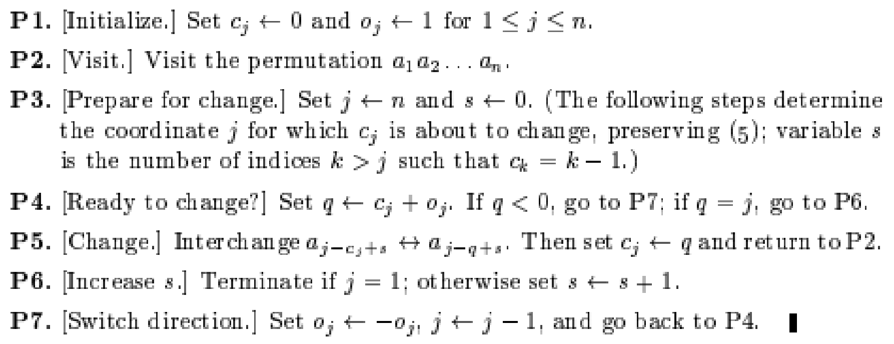
L3. [Increase a_j.] Set $l \leftarrow n$. If $a_j \geq a_l$, decrease l by 1 repeatedly until $a_j < a_l$. Then interchange $a_j \leftrightarrow a_l$. (Since $a_{j+1} \geq \cdots \geq a_n$, element a_l is the smallest element greater than a_j that can legitimately follow $a_1 \ldots a_{j-1}$ in a permutation. Before the interchange we had $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_l > a_j \geq a_{l+1} \geq \cdots \geq a_n$; after the interchange, we have $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_j > a_l \geq a_{l+1} \geq \cdots \geq a_n$.)

L4. [Reverse $a_{j+1} \ldots a_n$.] Set $k \leftarrow j + 1$ and $l \leftarrow n$. Then, if $k < l$, interchange $a_k \leftrightarrow a_l$, set $k \leftarrow k + 1$, $l \leftarrow l - 1$, and repeat until $k \geq l$. Return to L1. ▮

Figure 1: Knuth's description of Algorithm L [?]

the sequences efficiently to the user.

2. Algorithm P, plain changes, only requires the input sequence to contain distinct elements. Again, a new array of indices is created to avoid this issue arising.



P1. [Initialize.] Set $c_j \leftarrow 0$ and $o_j \leftarrow 1$ for $1 \leq j \leq n$.

P2. [Visit.] Visit the permutation $a_1 a_2 \ldots a_n$.

P3. [Prepare for change.] Set $j \leftarrow n$ and $s \leftarrow 0$. (The following steps determine the coordinate j for which c_j is about to change, preserving (5); variable s is the number of indices $k > j$ such that $c_k = k - 1$.)

P4. [Ready to change?] Set $q \leftarrow c_j + o_j$. If $q < 0$, go to P7; if $q = j$, go to P6.

P5. [Change.] Interchange $a_{j-c_j+s} \leftrightarrow a_{j-q+s}$. Then set $c_j \leftarrow q$ and return to P2.

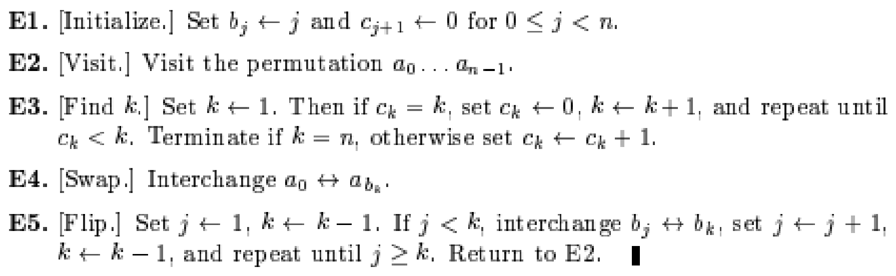
P6. [Increase s.] Terminate if $j = 1$; otherwise set $s \leftarrow s + 1$.

P7. [Switch direction.] Set $o_j \leftarrow -o_j$, $j \leftarrow j - 1$, and go back to P4. ▮

Figure 2: Knuth's description of Algorithm P [?]

Plain changes works by maintaining a second array, describing the direction that the elements in the sequence can "move". An element is mobile if it is pointing in the direction of one of its neighbours, and it's value is greater than its neighbour's. Each iteration, the largest mobile gets to move, creating a new sequence.

3. Algorithm E is similar to Algorithm P. It uses two arrays to control the next element swap that takes place.



E1. [Initialize.] Set $b_j \leftarrow j$ and $c_{j+1} \leftarrow 0$ for $0 \le j < n$.

E2. [Visit.] Visit the permutation $a_0 \ldots a_{n-1}$.

E3. [Find k.] Set $k \leftarrow 1$. Then if $c_k = k$, set $c_k \leftarrow 0$, $k \leftarrow k+1$, and repeat until $c_k < k$. Terminate if $k = n$, otherwise set $c_k \leftarrow c_k + 1$.

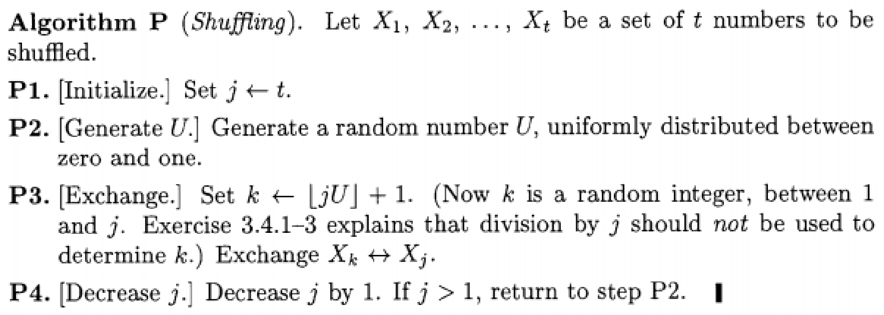
E4. [Swap.] Interchange $a_0 \leftrightarrow a_{b_k}$.

E5. [Flip.] Set $j \leftarrow 1$, $k \leftarrow k-1$. If $j < k$, interchange $b_j \leftrightarrow b_k$, set $j \leftarrow j+1$, $k \leftarrow k-1$, and repeat until $j \ge k$. Return to E2. ▮

Figure 3: Knuth's description of Algorithm E [?]

While Algorithm P only swapped elements that were adjacent, this always swaps the first element in the array to generate the next sequence. The additional control essentially constrains the swaps, which increases the efficiency of the algorithm.

4. The Fisher-Yates-Knuth Shuffle algorithm generates a random permutation of an input sequence [?][?].



Algorithm P (*Shuffling*). Let $X_1, X_2, \ldots, X_t$ be a set of t numbers to be shuffled.

P1. [Initialize.] Set $j \leftarrow t$.

P2. [Generate U.] Generate a random number U, uniformly distributed between zero and one.

P3. [Exchange.] Set $k \leftarrow \lfloor jU \rfloor + 1$. (Now k is a random integer, between 1 and j. Exercise 3.4.1–3 explains that division by j should *not* be used to determine k.) Exchange $X_k \leftrightarrow X_j$.

P4. [Decrease j.] Decrease j by 1. If $j > 1$, return to step P2. ▮

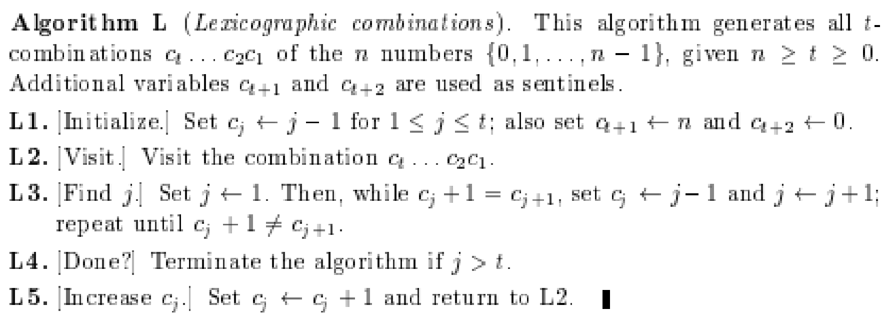
Figure 4: Description of Knuth Shuffles in *Seminumerical Algorithms* [?]

The algorithm requires a source of entropy to ensure its randomness. In the Python implementation, k is generated directly using the *random* library.

4.1.2 Combinations

Four algorithms for generating combinations were chosen to be evaluated.

1. The first algorithm is the straightforward once again. It proposes to visit all combinations in lexicographic order.



Algorithm L (*Lexicographic combinations*). This algorithm generates all t-combinations $c_t \ldots c_2c_1$ of the n numbers $\{0, 1, \ldots, n-1\}$, given $n \geq t \geq 0$. Additional variables c_{t+1} and c_{t+2} are used as sentinels.

L1. [Initialize.] Set $c_j \leftarrow j - 1$ for $1 \leq j \leq t$; also set $c_{t+1} \leftarrow n$ and $c_{t+2} \leftarrow 0$.

L2. [Visit.] Visit the combination $c_t \ldots c_2c_1$.

L3. [Find j.] Set $j \leftarrow 1$. Then, while $c_j + 1 = c_{j+1}$, set $c_j \leftarrow j - 1$ and $j \leftarrow j + 1$; repeat until $c_j + 1 \neq c_{j+1}$.

L4. [Done?] Terminate the algorithm if $j > t$.

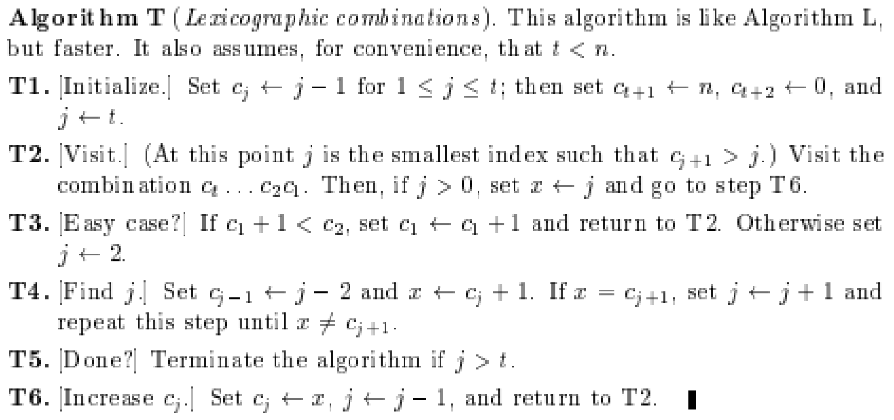
L5. [Increase c_j.] Set $c_j \leftarrow c_j + 1$ and return to L2. ▮

Figure 5: Description of Algorithm L [?]

Algorithm L finds the rightmost value which can still be increased. Once it has found and increased this value, it then reorders all previous elements to make the smallest value possible.

The description of the algorithm specifies constraints on the sequence that forms the combinations. This does not inhibit the practicality of the algorithm, as these numbers can be used as indices in the same way they were used for generating permutations.

2. Algorithm T is almost identical to Algorithm L. The difference occuring at stage T3. It identifies one of the most common situations that occurs when computing the next combination, and essentially hard-codes the best approach to take in that scenario.



Algorithm T (*Lexicographic combinations*). This algorithm is like Algorithm L, but faster. It also assumes, for convenience, that $t < n$.

T1. [Initialize.] Set $c_j \leftarrow j - 1$ for $1 \leq j \leq t$; then set $c_{t+1} \leftarrow n$, $c_{t+2} \leftarrow 0$, and $j \leftarrow t$.

T2. [Visit.] (At this point j is the smallest index such that $c_{j+1} > j$.) Visit the combination $c_t \ldots c_2 c_1$. Then, if $j > 0$, set $x \leftarrow j$ and go to step T6.

T3. [Easy case?] If $c_1 + 1 < c_2$, set $c_1 \leftarrow c_1 + 1$ and return to T2. Otherwise set $j \leftarrow 2$.

T4. [Find j.] Set $c_{j-1} \leftarrow j - 2$ and $x \leftarrow c_j + 1$. If $x = c_{j+1}$, set $j \leftarrow j + 1$ and repeat this step until $x \neq c_{j+1}$.

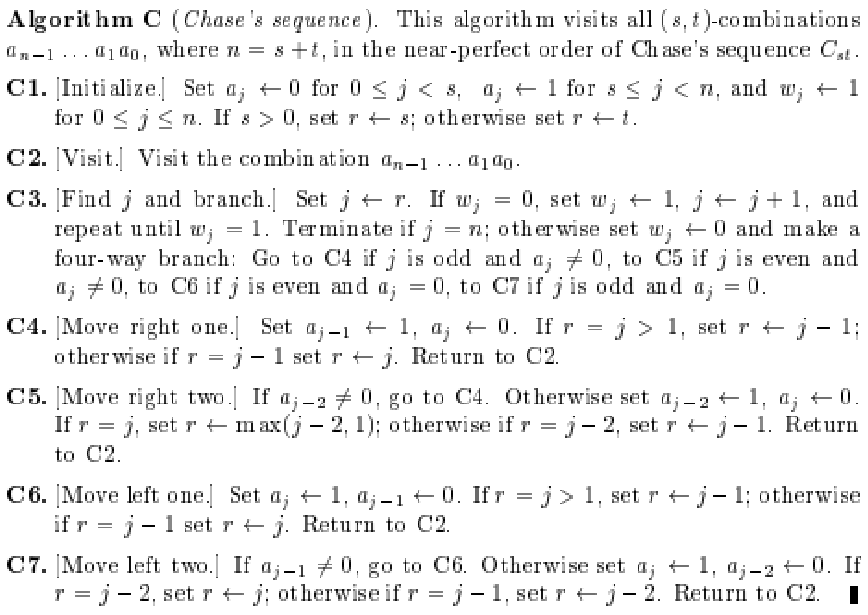
T5. [Done?] Terminate the algorithm if $j > t$.

T6. [Increase c_j.] Set $c_j \leftarrow x$, $j \leftarrow j - 1$, and return to T2. ▮

Figure 6: Description of Algorithm T [?]

Any minor losses at runtime due to the additional `if` statement are cancelled by the gains made when it returns `True`.

3. Algorithm C introduces the idea of near-perfect scheme [?]. They are characterized by the element in the sequence not having to move more than two indices away from it's current location to compute the next combination.



Algorithm C (*Chase's sequence*). This algorithm visits all (s,t)-combinations $a_{n-1}\dots a_1a_0$, where $n=s+t$, in the near-perfect order of Chase's sequence C_{st}.

C1. [Initialize.] Set $a_j \leftarrow 0$ for $0 \le j < s$, $a_j \leftarrow 1$ for $s \le j < n$, and $w_j \leftarrow 1$ for $0 \le j \le n$. If $s > 0$, set $r \leftarrow s$; otherwise set $r \leftarrow t$.

C2. [Visit.] Visit the combination $a_{n-1}\dots a_1a_0$.

C3. [Find j and branch.] Set $j \leftarrow r$. If $w_j = 0$, set $w_j \leftarrow 1$, $j \leftarrow j+1$, and repeat until $w_j = 1$. Terminate if $j = n$; otherwise set $w_j \leftarrow 0$ and make a four-way branch: Go to C4 if j is odd and $a_j \ne 0$, to C5 if j is even and $a_j \ne 0$, to C6 if j is even and $a_j = 0$, to C7 if j is odd and $a_j = 0$.

C4. [Move right one.] Set $a_{j-1} \leftarrow 1$, $a_j \leftarrow 0$. If $r = j > 1$, set $r \leftarrow j-1$; otherwise if $r = j-1$ set $r \leftarrow j$. Return to C2.

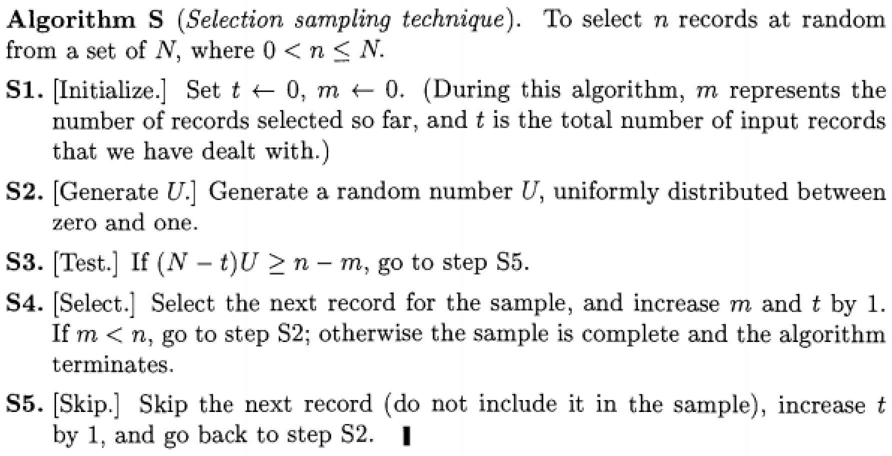
C5. [Move right two.] If $a_{j-2} \ne 0$, go to C4. Otherwise set $a_{j-2} \leftarrow 1$, $a_j \leftarrow 0$. If $r = j$, set $r \leftarrow \max(j-2, 1)$; otherwise if $r = j-2$, set $r \leftarrow j-1$. Return to C2.

C6. [Move left one.] Set $a_j \leftarrow 1$, $a_{j-1} \leftarrow 0$. If $r = j > 1$, set $r \leftarrow j-1$; otherwise if $r = j-1$ set $r \leftarrow j$. Return to C2.

C7. [Move left two.] If $a_{j-1} \ne 0$, go to C6. Otherwise set $a_j \leftarrow 1$, $a_{j-2} \leftarrow 0$. If $r = j-2$, set $r \leftarrow j$; otherwise if $r = j-1$, set $r \leftarrow j-2$. Return to C2. ▮

Figure 7: Description of Algorithm C [?]

4. Algorithm S employs a selection sampling technique for $\binom{n}{k}$. This works well when $k \leq n/2$.


Algorithm S (*Selection sampling technique*). To select n records at random from a set of N, where $0 < n \leq N$.

S1. [Initialize.] Set $t \leftarrow 0$, $m \leftarrow 0$. (During this algorithm, m represents the number of records selected so far, and t is the total number of input records that we have dealt with.)

S2. [Generate U.] Generate a random number U, uniformly distributed between zero and one.

S3. [Test.] If $(N - t)U \geq n - m$, go to step S5.

S4. [Select.] Select the next record for the sample, and increase m and t by 1. If $m < n$, go to step S2; otherwise the sample is complete and the algorithm terminates.

S5. [Skip.] Skip the next record (do not include it in the sample), increase t by 1, and go back to step S2. ▌

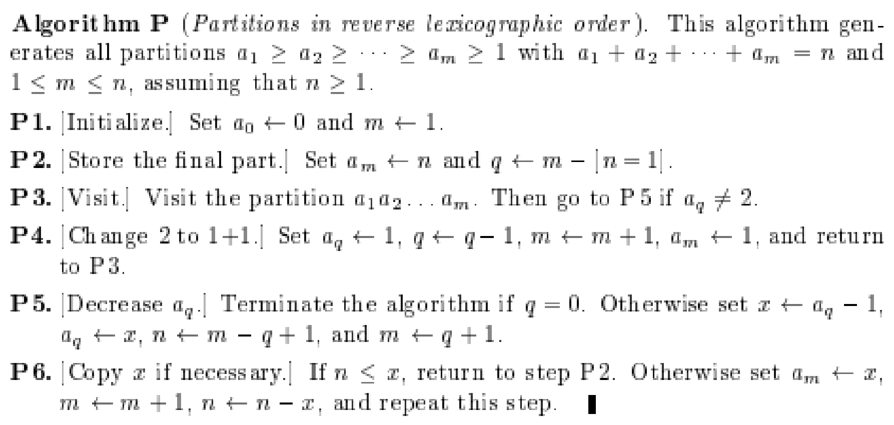
Figure 8: Algorithm S which generates a random combination [?]

However, there are improvements that can be made in the case of $k > n/2$. In this scenario, the problem turns from selection sampling to rejection sampling. The same process of selecting an element from the sequence takes place, but the element is then thrown away, rather than kept.

4.1.3 Partitions

Knuth did not cover a wealth of algorithms for partitioning in general. Two algorithms for integer partitioning were implemented.

1. The first algorithm generates all integer partitions for a given positive integer.



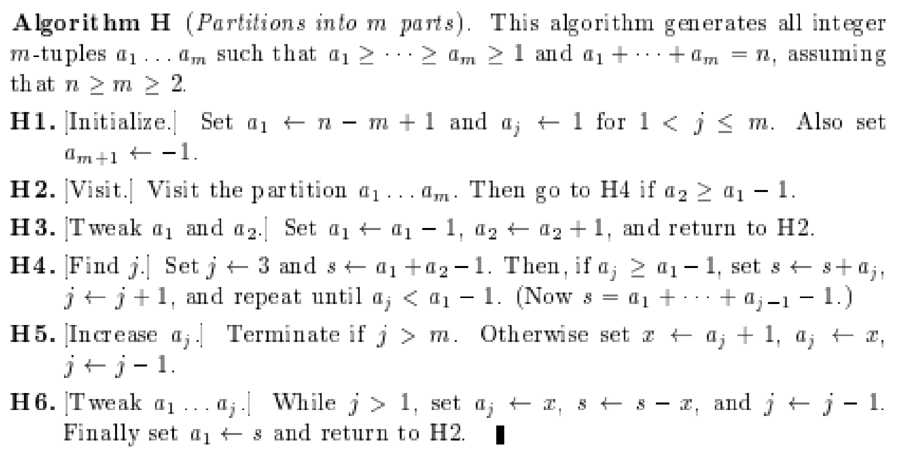
Algorithm P (*Partitions in reverse lexicographic order*). This algorithm generates all partitions $a_1 \geq a_2 \geq \cdots \geq a_m \geq 1$ with $a_1 + a_2 + \cdots + a_m = n$ and $1 \leq m \leq n$, assuming that $n \geq 1$.

```
P1. [Initialize.] Set $a_0 \leftarrow 0$ and $m \leftarrow 1$.
P2. [Store the final part.] Set $a_m \leftarrow n$ and $q \leftarrow m - [n = 1]$.
P3. [Visit.] Visit the partition $a_1 a_2 \ldots a_m$. Then go to P5 if $a_q \neq 2$.
P4. [Change 2 to 1+1.] Set $a_q \leftarrow 1$, $q \leftarrow q - 1$, $m \leftarrow m + 1$, $a_m \leftarrow 1$, and return
    to P3.
P5. [Decrease $a_q$.] Terminate the algorithm if $q = 0$. Otherwise set $x \leftarrow a_q - 1$,
    $a_q \leftarrow x$, $n \leftarrow m - q + 1$, and $m \leftarrow q + 1$.
P6. [Copy $x$ if necessary.] If $n \leq x$, return to step P2. Otherwise set $a_m \leftarrow x$,
    $m \leftarrow m + 1$, $n \leftarrow n - x$, and repeat this step. ▮
```

Figure 9: Algorithm P, for generating integer partitions [?]

Knuth discusses the key to the algorithm, which lies in P4 [?]. When a '2' is present in a partition, it is trivial to then find the following partition. The '2' is simply changed to a '1', and an additional '1' is appended to the end of the sequence.

2. Algorithm H returns the integer partitions of a given size in colexicographic order.



Algorithm H (*Partitions into m parts*). This algorithm generates all integer m-tuples $a_1 \ldots a_m$ such that $a_1 \geq \cdots \geq a_m \geq 1$ and $a_1 + \cdots + a_m = n$, assuming that $n \geq m \geq 2$.

H1. [Initialize.] Set $a_1 \leftarrow n - m + 1$ and $a_j \leftarrow 1$ for $1 < j \leq m$. Also set $a_{m+1} \leftarrow -1$.

H2. [Visit.] Visit the partition $a_1 \ldots a_m$. Then go to H4 if $a_2 \geq a_1 - 1$.

H3. [Tweak a_1 and a_2.] Set $a_1 \leftarrow a_1 - 1$, $a_2 \leftarrow a_2 + 1$, and return to H2.

H4. [Find j.] Set $j \leftarrow 3$ and $s \leftarrow a_1 + a_2 - 1$. Then, if $a_j \geq a_1 - 1$, set $s \leftarrow s + a_j$, $j \leftarrow j + 1$, and repeat until $a_j < a_1 - 1$. (Now $s = a_1 + \cdots + a_{j-1} - 1$.)

H5. [Increase a_j.] Terminate if $j > m$. Otherwise set $x \leftarrow a_j + 1$, $a_j \leftarrow x$, $j \leftarrow j - 1$.

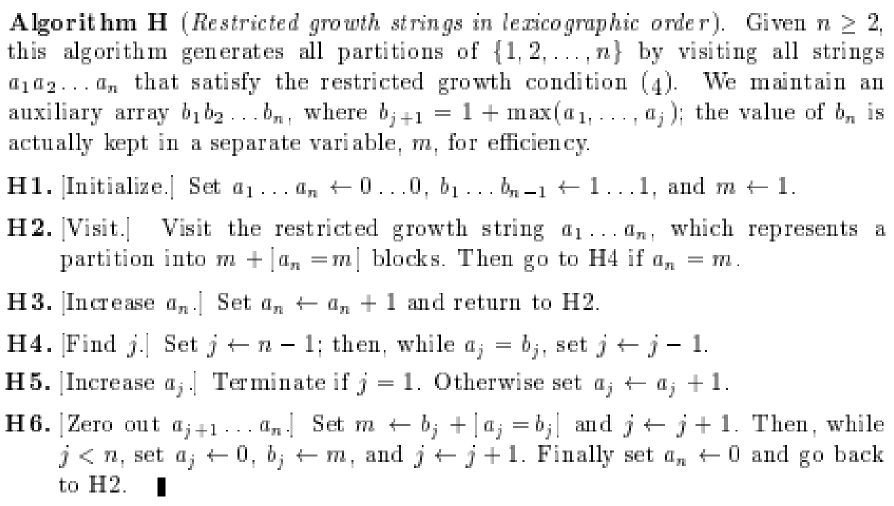
H6. [Tweak $a_1 \ldots a_j$.] While $j > 1$, set $a_j \leftarrow x$, $s \leftarrow s - x$, and $j \leftarrow j - 1$. Finally set $a_1 \leftarrow s$ and return to H2. ▮

Figure 10: Algorithm H, for generating integer partitions of m parts [?]

Despite Algorithm H being a niche case of Algorithm P, it may be worth investigating their performances when Algorithm H is called time and again in order to generate all possible integer partitions.

Knuth provided one algorithm for set partitioning, and another for multiset partitioning. External research outside the scope of Knuth's work was undertaken so that the package had the capability of generating random set partitions.

1. By applying the previous idea of creating a new array representing indices, the decision was taken to only implement the algorithm for generating all set partitions. Using the indices would allow multisets to be passed in as arguments, and the algorithm would be able to handle it adequately.



Algorithm H (*Restricted growth strings in lexicographic order*). Given $n \geq 2$, this algorithm generates all partitions of $\{1, 2, \ldots, n\}$ by visiting all strings $a_1 a_2 \ldots a_n$ that satisfy the restricted growth condition (4). We maintain an auxiliary array $b_1 b_2 \ldots b_n$, where $b_{j+1} = 1 + \max(a_1, \ldots, a_j)$; the value of b_n is actually kept in a separate variable, m, for efficiency.

H1. [Initialize.] Set $a_1 \ldots a_n \leftarrow 0 \ldots 0$, $b_1 \ldots b_{n-1} \leftarrow 1 \ldots 1$, and $m \leftarrow 1$.

H2. [Visit.] Visit the restricted growth string $a_1 \ldots a_n$, which represents a partition into $m + [a_n = m]$ blocks. Then go to H4 if $a_n = m$.

H3. [Increase a_n.] Set $a_n \leftarrow a_n + 1$ and return to H2.

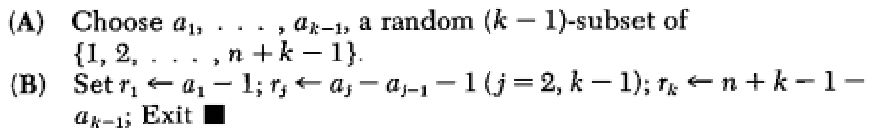
H4. [Find j.] Set $j \leftarrow n - 1$; then, while $a_j = b_j$, set $j \leftarrow j - 1$.

H5. [Increase a_j.] Terminate if $j = 1$. Otherwise set $a_j \leftarrow a_j + 1$.

H6. [Zero out $a_{j+1} \ldots a_n$.] Set $m \leftarrow b_j + [a_j = b_j]$ and $j \leftarrow j + 1$. Then, while $j < n$, set $a_j \leftarrow 0$, $b_j \leftarrow m$, and $j \leftarrow j + 1$. Finally set $a_n \leftarrow 0$ and go back to H2. ▮

Figure 11: Algorithm H, for generating all set partitions of a sequence [?]

2. It is important to understand that this algorithm was designed for the *Fortran* programming language, a verbose, low-level language. Python's strong list manipulation capabilities allow the algorithm to be implemented more smoothly with some minor adjustments.



(A) Choose $a_1, \ldots, a_{k-1}$, a random $(k-1)$-subset of $\{1, 2, \ldots, n+k-1\}$.
(B) Set $r_1 \leftarrow a_1 - 1$; $r_j \leftarrow a_j - a_{j-1} - 1$ $(j = 2, k-1)$; $r_k \leftarrow n + k - 1 - a_{k-1}$; Exit ■

Figure 12: Nijenhuis & Herbert's algorithm for k random partitions of a set. [?]

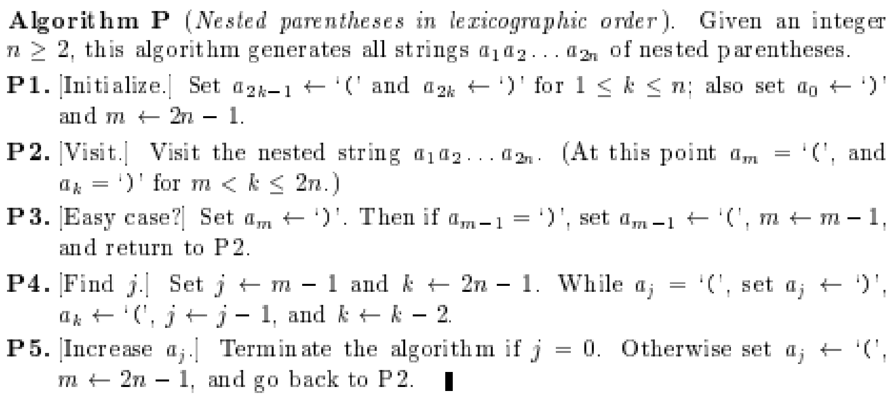
r represents the array containing the n elements of the input set. The reason it has more than n elements is that it needs to be able to represent where the partitions occur as unique elements in the set.

Due to list slicing in Python, there is no need to make a longer list, since they are mutable and the values can be easily changed. The first step remains the same; randomly choose where the partitions will occur. The implementation only differs in part 2, where the list is directly altered in Python to give an output which aligns with Algorithm H for set partitioning.

4.1.4 Trees

Three algorithms were implemented in Python for tree generation.

1. Algorithm P generates all trees of size n, represented as nested parentheses.



Algorithm P (*Nested parentheses in lexicographic order*). Given an integer $n \geq 2$, this algorithm generates all strings $a_1 a_2 \ldots a_{2n}$ of nested parentheses.

P1. [Initialize.] Set $a_{2k-1} \leftarrow$ '(' and $a_{2k} \leftarrow$ ')' for $1 \leq k \leq n$; also set $a_0 \leftarrow$ ')' and $m \leftarrow 2n-1$.

P2. [Visit.] Visit the nested string $a_1 a_2 \ldots a_{2n}$. (At this point $a_m =$ '(', and $a_k =$ ')' for $m < k \leq 2n$.)

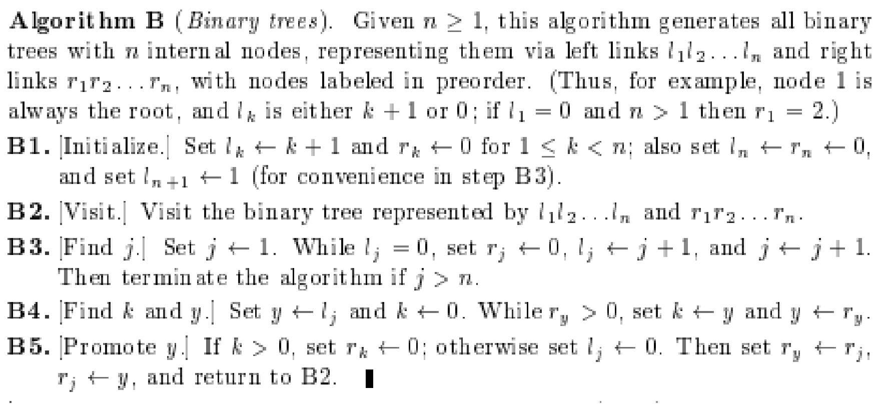
P3. [Easy case?] Set $a_m \leftarrow$ ')'. Then if $a_{m-1} =$ ')', set $a_{m-1} \leftarrow$ '(', $m \leftarrow m-1$, and return to P2.

P4. [Find j.] Set $j \leftarrow m-1$ and $k \leftarrow 2n-1$. While $a_j =$ '(', set $a_j \leftarrow$ ')', $a_k \leftarrow$ '(', $j \leftarrow j-1$, and $k \leftarrow k-2$.

P5. [Increase a_j.] Terminate the algorithm if $j = 0$. Otherwise set $a_j \leftarrow$ '(', $m \leftarrow 2n-1$, and go back to P2. ▮

Figure 13: Description of Algorithm P [?]

2. Algorithm B works specifically for binary trees. Rather than representing the structure using nested parentheses, the algorithm maintains two arrays which describe the left and right links between the nodes.



Algorithm B (*Binary trees*). Given $n \geq 1$, this algorithm generates all binary trees with n internal nodes, representing them via left links $l_1 l_2 \ldots l_n$ and right links $r_1 r_2 \ldots r_n$, with nodes labeled in preorder. (Thus, for example, node 1 is always the root, and l_k is either $k+1$ or 0; if $l_1 = 0$ and $n > 1$ then $r_1 = 2$.)

B1. [Initialize.] Set $l_k \leftarrow k+1$ and $r_k \leftarrow 0$ for $1 \leq k < n$; also set $l_n \leftarrow r_n \leftarrow 0$, and set $l_{n+1} \leftarrow 1$ (for convenience in step B3).

B2. [Visit.] Visit the binary tree represented by $l_1 l_2 \ldots l_n$ and $r_1 r_2 \ldots r_n$.

B3. [Find j.] Set $j \leftarrow 1$. While $l_j = 0$, set $r_j \leftarrow 0$, $l_j \leftarrow j+1$, and $j \leftarrow j+1$. Then terminate the algorithm if $j > n$.

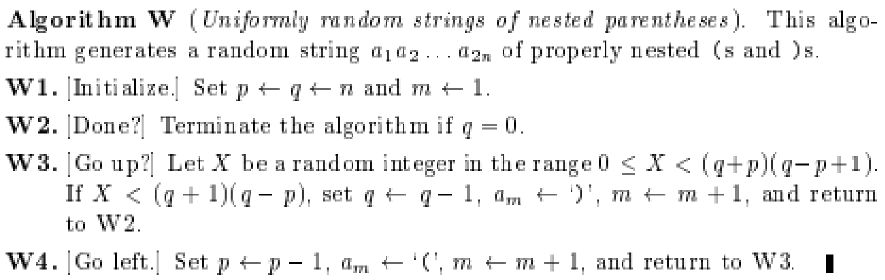
B4. [Find k and y.] Set $y \leftarrow l_j$ and $k \leftarrow 0$. While $r_y > 0$, set $k \leftarrow y$ and $y \leftarrow r_y$.

B5. [Promote y.] If $k > 0$, set $r_k \leftarrow 0$; otherwise set $l_j \leftarrow 0$. Then set $r_y \leftarrow r_j$, $r_j \leftarrow y$, and return to B2. ▮

Figure 14: Description of Algorithm B [?]

This representation creates some confusion as it is not as intuitive as the nested parentheses model. Furthermore, it also creates an issue of morphing it into a tree object.

3. Algorithm W offers the functionality of generating random trees of a given size. This algorithm returns trees as nested parentheses.



Algorithm W (*Uniformly random strings of nested parentheses*). This algorithm generates a random string $a_1a_2 \ldots a_{2n}$ of properly nested (s and)s.

W1. [Initialize.] Set $p \leftarrow q \leftarrow n$ and $m \leftarrow 1$.

W2. [Done?] Terminate the algorithm if $q = 0$.

W3. [Go up?] Let X be a random integer in the range $0 \le X < (q+p)(q-p+1)$. If $X < (q+1)(q-p)$, set $q \leftarrow q-1$, $a_m \leftarrow$ ')', $m \leftarrow m+1$, and return to W2.

W4. [Go left.] Set $p \leftarrow p-1$, $a_m \leftarrow$ '(', $m \leftarrow m+1$, and return to W3. ▮

Figure 15: Description of Algorithm W [?]

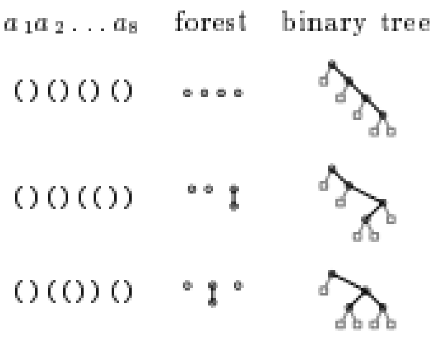


Figure 16: A partial table showing the relationship of nested parentheses to trees and forests. [?]

It is interesting to note the representation of the trees as nested parentheses. They can represent both a tree, and a forest as seen in figure 16. Custom parsers for the parentheses allow these interpretations. In contrast to this, the array of integers produced by Algorithm B is not as straightforward to manage. Any benefits made in the specific nature of the algorithm are lost when translating the output to a graph.

4.2 Experiments

4.2.1 Testing Generator Algorithms

The algorithms should be tested with varying lengths of input sequences. By calculating the mean time it takes to transform one valid combinatorial object to another, the expectation is that this value should remain similar regardless of the length of the input sequence.

The experiments are be carried out in Python using the *timeit* library to generate accurate times, and the results are plotted using Excel and the *matplotlib* Python library. In the cases of permutations and combinations, the basic Python algorithms will be tested against the *Cython* attempts, as well as *itertools'* capabilities.

It is important to acknowledge the shortcomings of time-based experiments when assessing algorithms. Any times found will be intrinsically linked to the hardware that the experiments are run on. However, since the goal is simply to find the 'quickest' algorithm to complete the task, this is still achievable since all algorithms will be gated by the same limitations.

4.2.2 Testing Random Algorithms

This project does not attempt to formally verify the randomness of its algorithms. They have already been proven, and as such, this randomness needs to be verified to ensure that this property was not lost in the Python implementations.

The $p - value$ returned by the chi-squared test in this case represents the probability that the actual sequence of occurs, on the condition that expected frequencies are, in fact, uniformly distributed. The $p - value$ returned is expected to be a relatively small value, given the number of possible partitions of 100,000 trials. Regardless, it should still be clear that the randomness of the algorithms is verified by this hypothesis test.

SciPy is a Python library which implements the chi-squared test [?]. Only one argument is required as input, the observed frequencies. It assumes a uniform distribution of the expected frequencies to create the null hypothesis. A double bar graph is presented to give a visual aid for the results.

4.3 Incomplete Work

4.3.1 Combinations - Algorithm T

Algorithm T for generating combinations [?] was not successfully implemented using Knuth's description. The current version of the algorithm in Python visits sequences in the wrong order, and duplicates values in the sequence erroneously.

In it's current state, it is inadequate and does not serve its purpose. Its inclusion in the package is revoked, and no tests will be carried out the incorrect algorithm.

4.3.2 Tree Parsing

In their current states, Algorithm P and Algorithm W return sequences of nested parentheses as mentioned above. Presently, the onus is on the user to decide the interpretation of this sequence, either as a binary tree or as a forest.

In future iterations of this package, it would be advantageous to have these functions take in an additional value, which would specify the return type of the sequences. The functionality for tree or forest parsing would come bundled with the package.

5 Evaluation

5.1 Appraisal of Completed System

The package meets the most expectations set out at the beginning. It contains methods that facilitate the generation of permutations, combinations, partitions and graphs. At its core, it is a developer-friendly package, with many obvious uses. The package acts as a centralized base for the most common combinatorial tasks.

Initially, the idea of graph theory was teased. Graph theory is an area of combinatorics which features widely across computer science, from networking to real-world maps. Unfortunately, due to time restrictions, graph theory had to be excluded from the scope of this project.

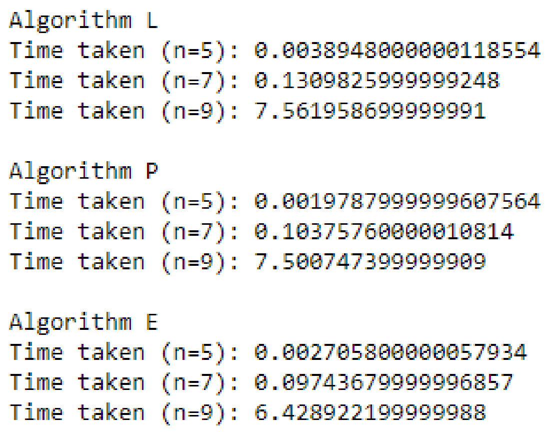
5.2 Testing the Package

Basic unit tests were setup to ensure that the algorithms provided the desired output for randomly selected input sequences. The amount of objects generated are tallied so that they match with the correct amount. By storing all the objects in a list, the list can be converted to a set, where any duplicates will be removed. Should the length of the list not equal the length of the set, it is clear that the algorithm is not functioning properly as it is outputting duplicated values.

5.2.1 Permutations

The three algorithms for generating all permutations of a sequence, mentioned in section 4.1.1, were timed using the *timeit* library. The total time taken to generate all of the permutations can be seen in figure 17, where n represents the size of the input sequence.

All algorithms perform similarly for smaller values of n, but the difference



```
Algorithm L
Time taken (n=5): 0.0038948000000118554
Time taken (n=7): 0.1309825999999248
Time taken (n=9): 7.561958699999991

Algorithm P
Time taken (n=5): 0.0019787999999607564
Time taken (n=7): 0.10375760000010814
Time taken (n=9): 7.500747399999909

Algorithm E
Time taken (n=5): 0.0027058000000057934
Time taken (n=7): 0.09743679999996857
Time taken (n=9): 6.428922199999988
```

Figure 17: Total time taken to generate all objects for some values of n.

quickly becomes apparent as n grows. This is visible when analysing the the graph of the mean time taken to generate the next permutation.

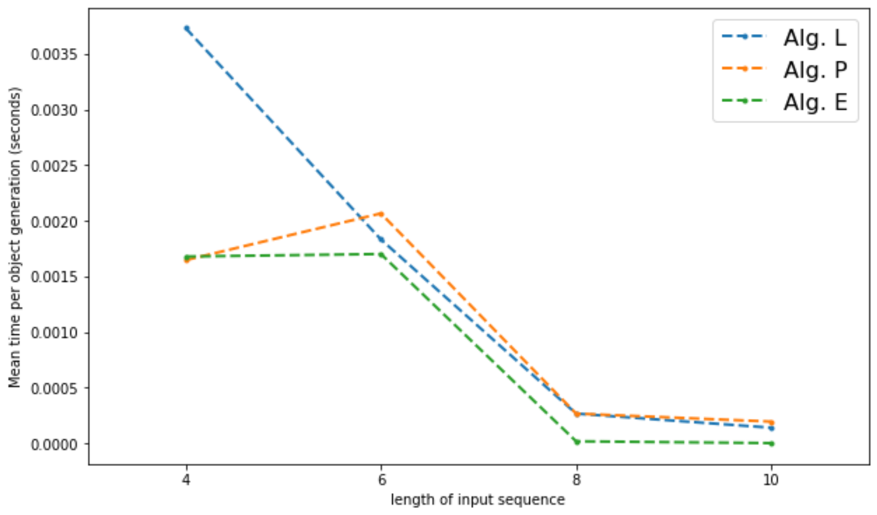


Figure 18: Mean time to generate a permutation for some values of n.

Algorithm E outperforms the other two for almost all values of n that were tested. Interestingly, the mean time decreases as n increases. This may appear counter-intuitive, given that the algorithms need to loop over a larger array. There are several reasons for this decrease. The overhead of initializing the arrays and variables at the start of the algorithm is distributed across all permutations, decreasing the average time. As n increases, the most common steps for generating the next object occur more and more often. Combine this with the increased efficiency of caching in memory when using lists in Python, this accounts for the overall decrease in the mean time.

Algorithm E was selected as the best performer based on the above results, and as such was implemented using *Cython*. After calculating the mean times for both Algorithm E and *itertools*, the following results were observed:

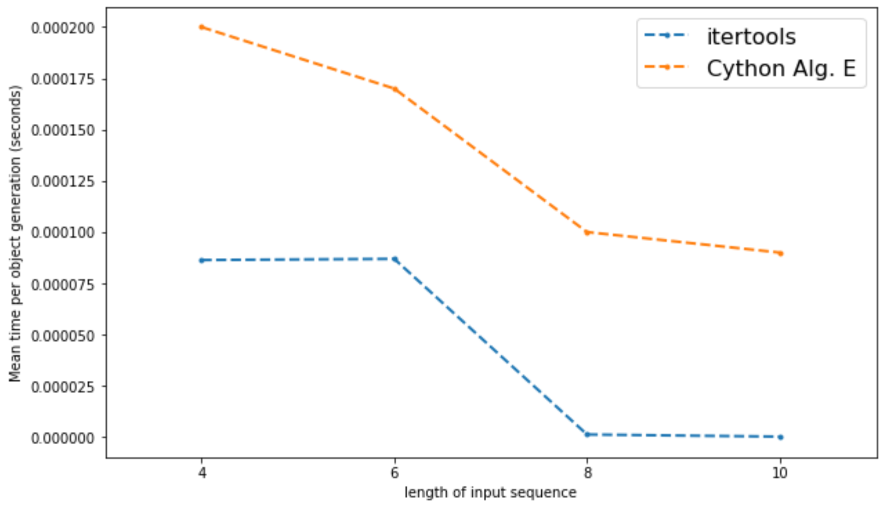


Figure 19: Algorithm E in Cython vs *itertools* for some values of n.

Itertools wins out against Algorithm E. We do see marked improvements in Algorithm E, however, compared to the default Python implementation. While Algorithm E was never going to match *itertools* due to its implementation in pure C, it does offer a competitive and viable alternative.

For the random permutation algorithm [?], the following frequencies were observed for 100,000 iterations of the algorithm.

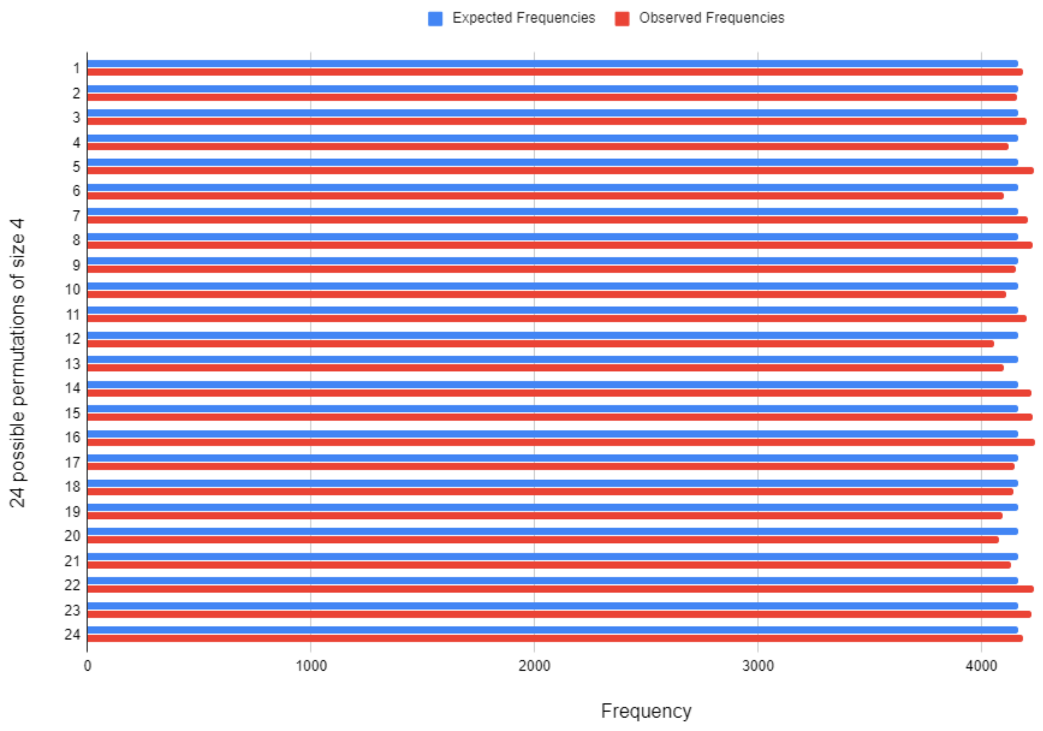


Figure 20: Expected vs Observed Frequencies

The observed frequencies tend to close match the actual frequencies. Performing the chi-squared test on the observed frequencies returns $p - value = 0.64$. This is relatively high, and states that given a uniform distribution, the probability of seeing the observed frequencies is approximately 64%. This is not enough to reject the null hypothesis, affirming the randomness of the algorithm.

5.2.2 Combinations

The same process used for permutations is repeated here. The size of the combinations being generated was locked to four, and the length of the sequences then vary to change the amount of combinations that are being generated.

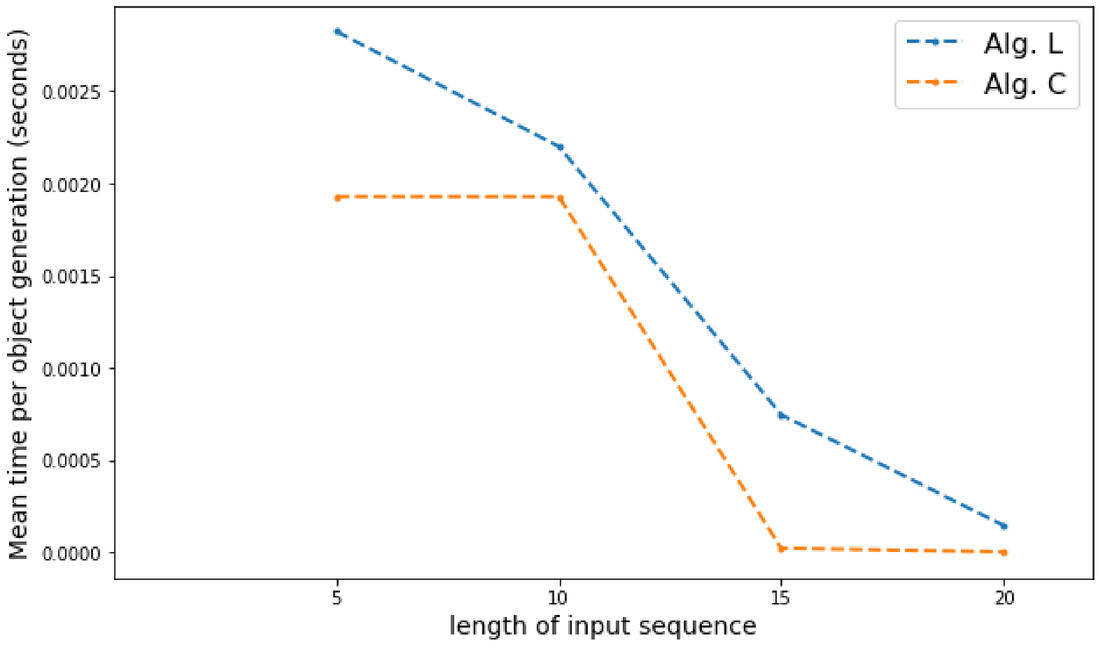


Figure 21: Mean time to generate a combination for some values of n

These algorithms follow a similar trend to the algorithms for permutations. Algorithm C boasts favourable results for all lengths of input sequences. The mean time decreases as n increases for the same reasons specified in the previous section. Algorithm C was enhanced with *Cython* and compared to *itertools*.

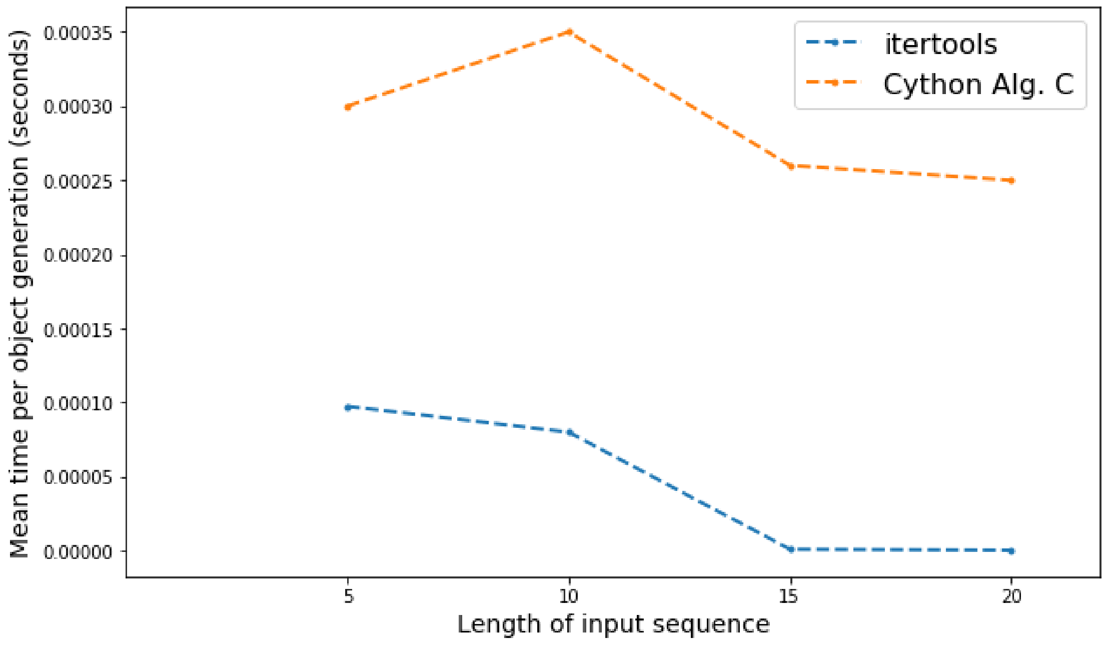


Figure 22: Evaluation *itertools* combination method to Algorithm C [?]

The *Cython* implementation of Algorithm C has a significant performance increase, but not enough to match *itertools*. It is not quite as competitive as the package's permutation generator, but it generates combinations at a sufficient rate.

The frequencies were recorded for the 21 combinations of $\binom{7}{5}$, as a similar amount of outcomes is favourable to keep the experiments fair and unbiased. Figure 23 illustrates a plausible random distribution of frequencies.

The chi-squared test null hypothesis for this case states that the frequencies are independent, and maintain no relation to each other. The $p - value = 0.35$ was obtained by running the test on the observed combination frequencies. While lower than the $p - value$ for permutations, it is still significantly higher than the required $p - value = 0.05$ to reject the null hypothesis. Thus, it can be said with some degree of certainty that this implementation produces suitably random results.

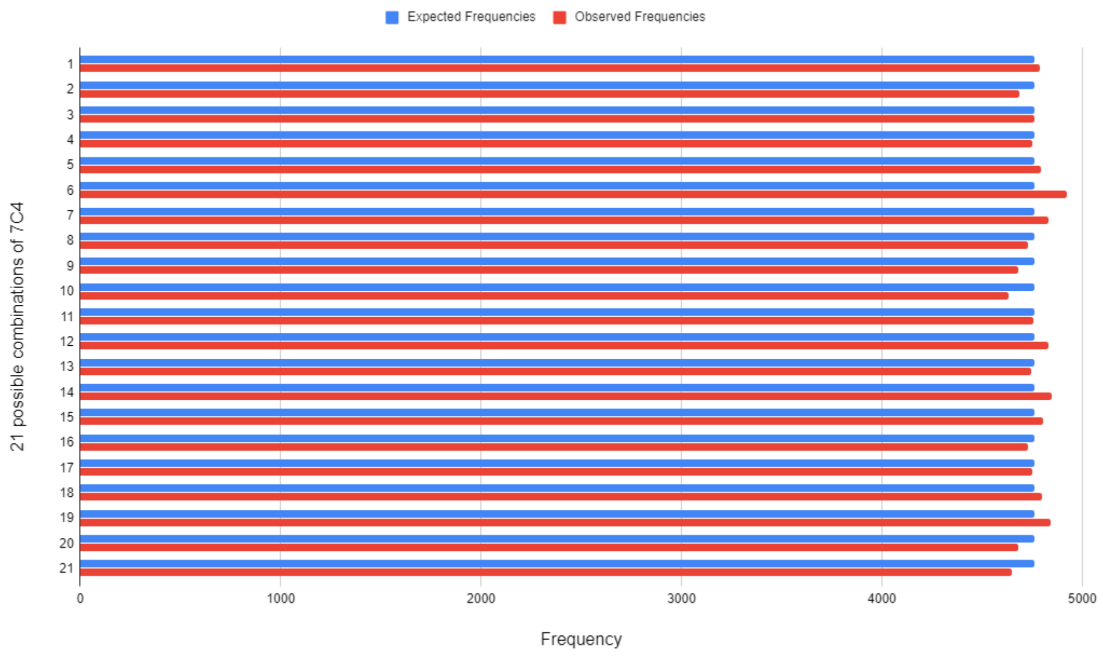


Figure 23: Frequency distribution of random combinations [?]

For the sake of completeness, the experiment was reran for $\binom{7}{3}$. This was to verify the the improvement made by utilising an enhancement suggested by Knuth, discussed earlier in section 4.1.2 [?]. That is to say, both selection sampling and rejection sampling produce random results. The $p - value$ obtained was 0.63, which reiterates the assertion that the algorithm is random.

5.2.3 Partitions

Apart from testing the algorithms to verify that they produced all the correct outputs, there was little comparison to be done due to the lack of variety. One of the test that was carried out, was whether Algorithm H [?] could produce the same outputs and Algorithm P [?] when wrapped in a `for` loop to specify the varying amount of partitions.

The resulting graph nicely illustrates the exponential increase in the number

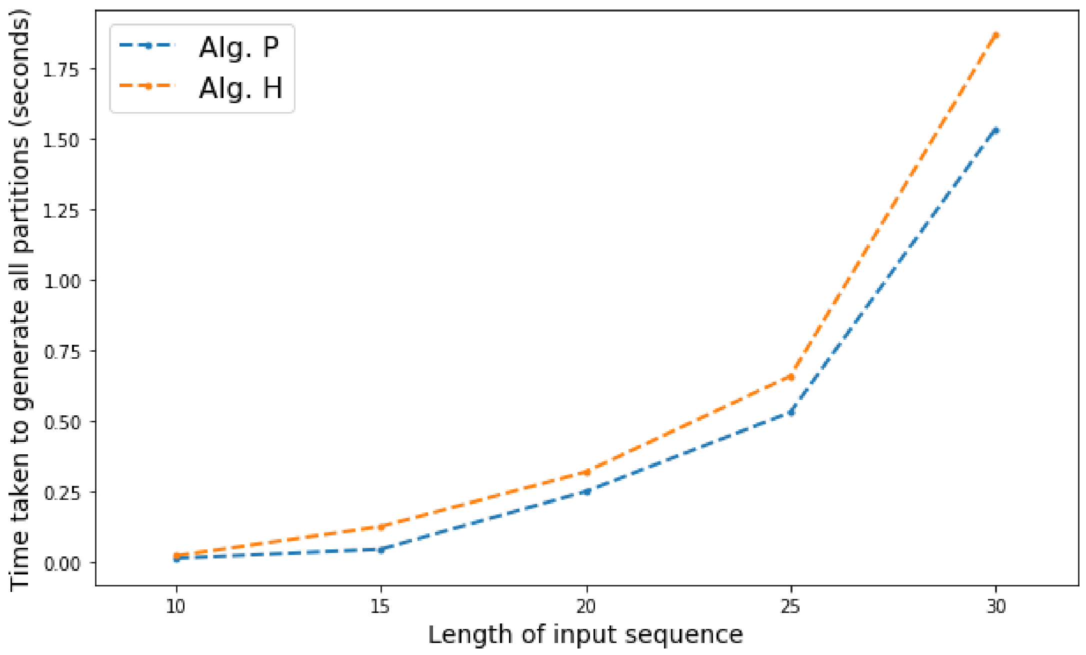


Figure 24: A comparison between Algorithm P [?] and Algorithm H [?] for integer partitioning

of integer partitions, as the integer increases. It also exposes the efficiency of Algorithm P for generating all partitions of differing sizes.

Nijenhuis & Herbert's algorithm for set partitioning [?] must have its randomness verified by the chi-squared test. The algorithm specifies the number of partitions that the sequence should be divided into, so the choice was made to generate all 3-partitions of a set containing ten elements, give 36 unique partitions.

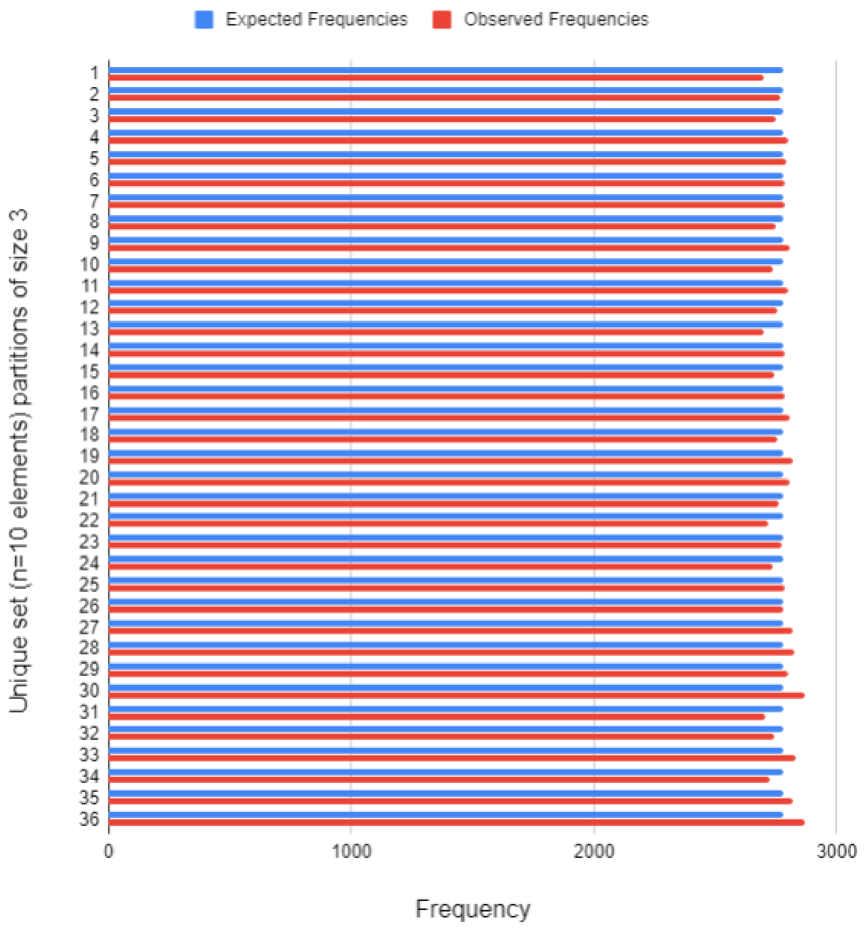


Figure 25: The expected and observed frequencies of a random set partitioning algorithm [?]

The chi-squared test verified the functions randomness, returning $p-value = 0.93$. This overwhelmingly supports the null hypothesis, that there is no relationship between the set partitions.

5.2.4 Trees

Graphing the distribution of frequencies for Algorithm W [?] uncovered undesirable results.

Visually, the distribution appears biased, with many observed frequencies

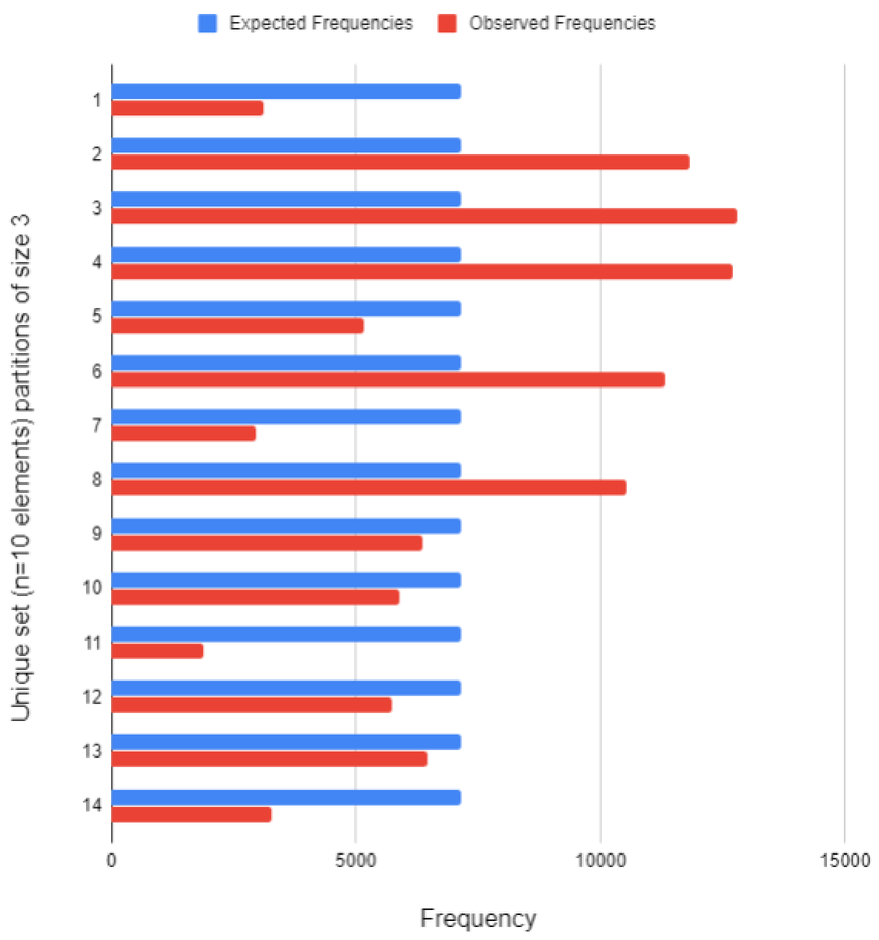


Figure 26: The expected and observed frequencies of a random tree generation algorithm

that have tallies distant from the expected. This observation is supported by the chi-squared test, calculating $p - value = 0.0$, which unequivocally rejects the null hypothesis at a 5% significance level. Evidently, Algorithm W does not generate uniformly random strings of nested parentheses, despite generating all of them at multiple times.

6 Conclusion

The result of the project is a multipurpose package with the ability to generate the most common combinatorial objects. It enables developers to quickly access tools they need, instead of pulling resources from multiple different libraries. The project gives a voice to algorithms which otherwise may have been overlooked. Repeatedly, interesting and unexpected results cropped up.

Knuth's algorithms performed comparably to *itertools'* implementations of permutations and combinations. There is still room for improvement to be made here, which will explored in section 6.1. However, not all outcomes were as favourable as these, specifically Algorithm W for generating random trees. In spite of this hiccup, the other algorithms for generating random combinatorial objects all produced acceptably random results.

The work completed here is almost entirely based on existing knowledge. In *The Art of Computer Programming Volume 1: Fundamental Algorithms*, Knuth says "an algorithm must be seen to be believed." This project supports his declaration, by giving a vision to algorithms, and providing competition to current standard. Competition is the greatest means of improvement and innovation.

6.1 Future Work

The area which immediately requires the most attention is the generation of random combinatorial objects. This package does not currently support such a method. The issue surely lies in the translation to Python, but it is not clear exactly where this occurs. On brief examination, the problem appears to lie with the source of entropy, and how the entropy is employed to attain uniform randomness. This is evident due to the correct objects appearing when the algorithm is employed sufficiently at least once. Additional work in this area should be focused on correcting what is a likely bug in the implementation, or researching new methods of random tree generation outside the scope of *The Art of Computer Programming Volume 4A: Combinatorial Algorithms.*

Another area demanding attention is in the *Cython* implementations of Algorithm E [?] and Algorithm C [?]. Rudimentary changes were made to the algorithms, offering significant improvement. The key to unlocking better performance lies in the use of data structures in the algorithms. Surprisingly, performance decreased after implementing C arrays instead of Python. As a result, these changes were withdrawn. The poor performance is likely due to the clumsy array manipulation in C, which led to shoddy loops over the arrays to change values. A promising solution to this employs *numpy* arrays to act as the array structures. *Numpy* allows direct accessing of the data buffer at C speed, while maintaining better functionality supporting manipulation [?].

6.2 How the Project was Conducted

The project was divided into the four key problem areas: permutation, combinations, partitions, and trees. They were addressed one by one. The work started on permutations, in October 2019. Reading through Knuth's book was the initial step, where new concepts and ideas were introduced. Work began on implementing some of these algorithms in Python.

Come November 2019, the project moved on to look at combinations. The same approach was taken once again, and by the end of month, a package was beginning to take shape. There were now several algorithms which facilitated the generation of permutations and combinations. Work then slowed due to the looming exam season, before briefly continuing in late December and early January. Here, *Cython* was explored. The package provided some competition to other packages.

Returning to college, the focus changes to trees. Growing more familiar with Knuth's work, the algorithms and ideas became easier to grasp. February saw partitions being explored, and by the end of the month, algorithms existed to offer functionality in each area. As the open-day approached, popular uses of the algorithms were investigated. This quickly ended however, due to the shutdown of the college. From here, work continued from home, attempting to shape the package further.

6.3 Reflection

The primary takeaway from this project is the effort required in order produce programming library of some substance. Before writing a single line of code, countless hours of research are needed in order to develop a satisfactory level of knowledge of an area. It is crucial to understand both theoretical and applied material that already exists in the problem space, to avoid repeating unnecessary work. Developing the project gave a fascinating insight into the tree-like structure that is the theoretical material. It was incredible to be able to trace ideas back through time, and relive the same realisations these great minds had before they wrote about them, all while nurturing an understanding of a momentous area of computer science.

The process of developing the package promoted increased literacy in Python. Constant manipulation of lists in Python led to a greater understanding of the underlying data structures. Improvements were perpetually chased, aiding an ever-growing knowledge of the intricacies of Python.

It would be unfair not to mention the global pandemic caused by the COVID-19 virus. Reflecting back on early March, there was no way to predict the disruption it would cause. Home suddenly became the only working environment available; not an ideal situation. In difficult and trying times, we turn to our friends and family for assistance. COVID-19 ripped this comfort away.

The virus highlights the need for proper time-management. The future is unpredictable, and time is not a guarantee. Upon reflection, progress on this report should have began sooner. Although ideas and notes were kept from the beginning, the circumstances that arose have been detrimental. A more favourable approach would have been to start developing the report possibly as early as January, when the majority of the work on permutations and combinations was already complete.

References

[1] Chi-squared statistical hypothesis test in *SciPy* package for Python. URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chisquare.html`

[2] Cython static compiler for Python. URL: `https://cython.org/`

[3] Fisher, Ronald A.; Yates, Frank. *Statistical tables for biological, agricultural and medical research (3rd ed.)* (1948) [1938], pp. 26–27

[4] *itertools* Python package. URL: `https://docs.python.org/3/library/itertools.html`

[5] *itertools* Python package extension tools. `https://docs.python.org/3/library/itertools.html#itertools-recipes`

[6] Knuth, Donald. The Art of Computer Programming Volume 2, *Seminumerical Algorithms* 3.4.2 (1998), pp. 142-143.

[7] Knuth, Donald. The Art of Computer Programming Volume 2, *Seminumerical Algorithms* 3.4.2 (1998), pp. 145.

[8] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.2 (2005), pp. 1-2. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc2b.pdf`

[9] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.2 (2005), pp. 4. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc2b.pdf`

[10] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.2 (2005), pp. 19-20. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc2b.pdf`

[11] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.2 (2005), pp. 11. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc2b.pdf`

[12] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.3 (2005), pp. 4-5. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc3a.pdf`

[13] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.3 (2005), pp. 5. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc3a.pdf`

[14] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.3 (2005), pp. 11-15. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc3a.pdf`

[15] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.4 (2005), pp. 2. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc3b.pdf`

[16] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.4 (2005), pp. 2-3. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc3b.pdf`

[17] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.5 (2005), pp. 26-27. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc3b.pdf`

[18] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.6 (2005), pp. 2-3. URL: `http://www.cs.utsa.edu/~wagner/knuth/fasc4a.pdf`

[19] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.6 (2005), pp. 4-5. URL: http://www.cs.utsa.edu/~wagner/knuth/fasc4a.pdf

[20] Knuth, Donald. The Art of Computer Programming Volume 4, *Combinatorial Algorithms* 7.2.1.6 (2005), pp. 13. URL: http://www.cs.utsa.edu/~wagner/knuth/fasc4a.pdf

[21] Nijenhuis, Albert; Herbert, Wilf. *Combinatorial Algorithms For Computers and Calculators 2nd Edition* (1978), pp. 52-53. URL: https://www.math.upenn.edu/~wilf/website/CombinatorialAlgorithms.pdf

[22] *NumPy* Python package. URL: https://cython.readthedocs.io/en/latest/src/userguide/numpy_tutorial.html#efficient-indexing-with-memoryviews

[23] Python Generator Functions. URL: https://wiki.python.org/moin/Generators