A Framework to Discover Combinatorial Algorithms

Rama Badrinath[1], Pramod Ganapathi[2], and Abhiram Natarajan[3]

1 **Amagi Media Labs, Bangalore, India.**
2 **Department of Computer Science, Stony Brook University, USA.**
3 **Department of Computer Science, Purdue University, USA.**

Abstract

We present a framework to discover combinatorial algorithms. The framework consists of a generic algorithmic structure built using backtracking. With simple tweaks to the structure, we are able to generate several combinatorial objects such as permutations, combinations, subsets, Catalan families, integer compositions, integer partitions, gray codes, derangements, palindromes, solutions to Diophantine equations with positive coefficients, solutions to sudoku, and solutions to n-queens problem. The algorithms are general and flexible in the sense that they work with several output constraints such as duplicate elements, restriction on the number of elements chosen at a time, and a limit on the number of times an element can be used.

1998 ACM Subject Classification G.2.1 Combinatorics

Keywords and phrases framework; combinatorial algorithms; permutations; combinations; subsets; diophantine; palindromes; partitions; compositions; count array

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Given 100 male actors and 150 female actors, in how many ways *formula movies* (with almost the same plot) can be produced having a hero, a heroine, and a villain? In how many ways 5 couples can stand next to each other for a group photo such that every husband-wife pair are not next to each other? How many words a hacker has to try in the worst-case to crack a password of length at most 16 using commonly-allowed letters? In how many ways a knock-out wresting tournament can be held among 65 wrestlers? These are some of the questions that can be answered in combinatorics. The algorithms to investigate solutions of combinatorial problems are called combinatorial algorithms.

Combinatorial algorithms explicitly or implicitly explore all possible solution instances from a large combinatorial universe and satisfy certain properties. At the core, almost all such algorithms generate combinatorial objects such as permutations, combinations, and subsets. They are used in innumerable application areas including software & hardware testing, molecular biology, cryptography, and operations research [17, 33].

A great multitude of algorithms have been presented over the years to generate combinatorial objects. Often, we see that there are significant similarities between these algorithms, leading us to the question of trying to find an elegant paradigm that is common to all of them. To this end, developing a generic algorithm or framework that could produce several combinatorial algorithms will be extremely useful for several reasons. First, it can be used to discover new combinatorial algorithms. Second, the understanding of the framework will aid in understanding how combinatorial objects are generated from different combinatorial algorithms. It would help us understand the fundamental atomic steps behind combinatorial generation. Third, it becomes easy to prove correctness of a combinatorial algorithm that can be discovered from the framework. Finally, and most importantly, it would help people preparing for software engineering interviews (aka future world changers) to save space in their heads, because they would have to memorize only this particular framework, and use the remaining space for other critical things such as remembering the best items on the closest restaurant's menu.

In this paper, we present a powerful framework that can be used to discover a wide variety of combinatorial algorithms. We use our framework to generate several combinatorial objects such as permutations, combinations, subsets, Catalan families, integer compositions, integer partitions, derangements, palindromes, gray codes, solutions to Diophantine equations with positive coefficients, solutions to sudoku, and solutions to n-queens problem. The framework is not only flexible and general, but robust as well. It is robust in the sense that it works with various output constraints such as objects with repeated (or duplicate) elements, objects taking not all but only a few elements at a time, and having a (lower and/or upper) limit on the number of times an element can be used.

Our framework is based on the backtracking algorithm design technique (see Levitin [27]). It gives a specific algorithmic structure for combinatorial generation. The framework constructs a state-space tree on-the-fly consisting of all feasible choices for every component of the combinatorial object. In the state-space tree, nodes with feasible choices for the first component are at level 1, nodes with feasible choices for the second component are at level 2, and so on. The options for each component are visited in a depth-first traversal order generating the combinatorial objects in the lexicographic order. If there are no legitimate choices left for a particular node, then the algorithm backtracks to its parent node and considers the next feasible option for the previous component. In this way combinatorial objects are generated efficiently pruning unnecessary subtrees.

The most complicated data structure that the framework uses is an array, we call count array, which simply stores the allowed number of repetitions of the various unique elements. Each time an instance of an element is used for a specific component of a combinatorial object, the count of the number of available instances is decremented (meaning non-availability of those instances). When we backtrack from a node to its parent node, the count of the number of available instances of the corresponding relevant component is incremented (meaning availability of those instances). This straightforward method of decrementing and incrementing in the data structure with problem-dependent constraint logic allows us to generate a variety of combinatorial objects.

Our contributions. The major contributions of this paper are:

(1) [Framework.] We present a framework built on backtracking that can be used to discover a wide variety of combinatorial algorithms.

(2) [Combinatorial algorithms.] Using the framework and elementary auxiliary data structures, we develop several algorithms for solving basic combinatorial problems. We remark that the algorithms have a remarkable amount of flexibility.

Organization of the paper. The outline of the paper is as follows. In Section 2, we present a framework to discover combinatorial algorithms. In Section 3, we present algorithms for problems such as generating permutations, combinations, subsets, etc., using our framework and other elementary data structures. Related work is given in Section 4.

2 Combinatorial Framework

In this section, we present a framework based on backtracking that can be used to discover combinatorial algorithms.

Notations & terminologies. We now introduce some notation that is used throughout the paper. The sorted list of input elements is L. The size of L is denoted by n and the number of unique (or distinct) elements in L is denoted by u. A list U contains the unique elements (in sorted order) of L. The list $\mathcal{R}$ contains a generated combinatorial object, which in turn is a list of components. This list is overwritten multiple times during the program execution. The term *index* refers to the various indices of $\mathcal{R}$ where components are populated. There is an array P such that $P[ele]$ denotes the position of the element ele in array U. The number of possible choices at a node is given by *max_choices*. The ith choice is represented by $choice_i$ and its cost is denoted by $cost_i$. The term $CA[i]$ generally represents the number of occurrences of $choice_i$. It might also mean the total cost depending on the problem.

Framework. Figure 1 gives a framework $\mathcal{F}$ to discover combinatorial algorithms. For simplicity, we assume that the variables $\mathcal{R}, CA, choice, cost$, and *max_choices* are all defined in the global scope.

```
F(index)
if REJECT($\mathcal{R}$, $index$) then
  return
elseif ACCEPT($\mathcal{R}$, $index$) then
  PRINT($\mathcal{R}$, $index$)
else
  for $i \leftarrow 1$ to $max\_choices$ do
    if $choice_i$ is feasible then
      $\mathcal{R}[index] \leftarrow choice_i$
      $CA[i] \leftarrow CA[i] - cost_i$
      $\mathcal{F}(index + 1)$
      $CA[i] \leftarrow CA[i] + cost_i$
```

Figure 1 Framework to discover combinatorial algorithms.

The function $\mathcal{F}$ builds or visits a combinatorial state-space tree of nodes in depth-first traversal order. Node with $index = 0$ is the root node. Nodes with $index = 1$ are at level 1, and so on. The term *max_choices* denotes the branch factor (or fan-out) of a node. The program control visits only those nodes that are feasible. When the control reaches a reject state node, it returns to its parent node. When the control is at an accept state node, then the algorithm prints the combinatorial object, which consists of the components along the path from the root to the present node.

The core idea of the framework comes from the logic that when $choice_i$ is feasible, it is assigned to the *index*-th component of $\mathcal{R}$. The $CA[i]$ value is decremented by $cost_i$ (which is often 1) to denote that the resource corresponding to $choice_i$ has depleted. In other words, the depletion has been accounted for. Then the algorithm searches for further components. If all choices are either infeasible or fully considered, then the algorithm backtracks after incrementing the $CA[i]$ value by $cost_i$ again to mean that the resources corresponding to $choice_i$ has become available.

Most existing combinatorial algorithms are based on one of the following methodologies: (a) ranking or unranking – defining a bijective function between natural numbers and combinatorial objects; (b) lexicographic order generation – dealing with the rightmost element of every instance; and (c) minimal change property – where the new object can be obtained from the previous object with little changes, e.g.: gray code. Our framework generates combinatorial objects in the lexicographic order as it always affects the last component of the partially computed combinatorial object and scans the choices in lexicographic order.

3 Combinatorial Algorithms

In this section, we present several combinatorial algorithms discovered from the framework elucidated in Section 2. The pseudocodes of all algorithms in this section are given in Figure 2. The recursion trees of PERMUTATIONS, COMBINATIONS, CATALAN, and COMPOSITIONS algorithms are given in Section A in Appendix.

Permutations. The PERMUTATIONS algorithm derived from the framework matches with the one proposed by Rohl [38] and has the following salient features: (*i*) generates permutations taking $r (\in [1, n])$ elements at a time, and (*ii*) generates permutations for a multiset (or set containing repeated elements).

We first build the count array CA that maintains the number of occurrences of each unique element in the input list. For example, if the input list (in sorted order) is $L = \langle a, b, b, b \rangle$, then $n = 4, u = 2, U = \langle a, b \rangle$, and $CA = \langle 1, 3 \rangle$. The PERMUTATIONS algorithm uses CA to effect and generates the required combinatorial objects. It is not hard to convert the recursive procedure to an iterative one.

To establish an upper bound on the running time of PERMUTATIONS, let us analyze the recursion tree shown in Figure 4. We see that the number of leaves are exactly equal to the number of unique r-permutations of L, i.e., $|\mathcal{P}^r(L)|$. Each of the leaves are exactly $(r + 1)$ levels below the root. The critical operations, i.e., looping and searching for available elements, are done on the first r levels.[1] At each level, we loop in the range $[1, u]$. Thus, the running time of PERMUTATIONS is bounded by $\mathcal{O}\left(|\mathcal{P}^r(L)| \times r \times u\right)$ in the worst case.

Combinations. One simple change to PERMUTATIONS algorithm can produce all possible r-combinations of an input list. Before assigning an element at a particular index in $\mathcal{R}$, we make sure that the element occupies a position in L that is greater than the position of element at the previous index. At the first index however, we may assign any element. The additional condition required is: $(index = 1$ **or** $i \geq P[\mathcal{R}[index - 1]])$. We use the condition $i > P[\mathcal{R}[index - 1]]$ to get combinations that contain no repeated elements.

Though the modified algorithm generates r-combinations correctly, with certain kinds of input there can be many wasteful branches i.e., recursion branches that do not produce any output. All unnecessary computations are because for the choices we loop in the range $U[1], \ldots, U[u]$ at every index. However, based on two simple observations, we can completely eliminate all wasteful branches: (*i*) Once $U[k]$ is assigned at a particular index of $\mathcal{R}$, subsequent indices can only contain one of $U[k + 1], \ldots, U[u]$, and (*ii*) In the case where we have already assigned elements to the first k indices of $\mathcal{R}$, and $\mathcal{R}[k] = U[y]$, we need not loop if we are sure that there are fewer than $(r - k)$ elements left over, i.e., if $\sum_{i=k}^{u} CA[i] < (r - k)$, we can terminate looping in the current recursion level and return to the previous recursion level.

The loop limits would need to be made tighter. For simplicity, we modify the definition of $P[ele]$ to return 0 if $ele \notin U$. Also, we assign to $\mathcal{R}[0]$ an element that does not exist in U. This would mean that $P[\mathcal{R}[0]] = 0$. Observation (*i*) indicates that we can begin looping from $P[\mathcal{R}[index - 1]]$ instead of 1. To incorporate Observation (*ii*), we build a cumulative count array, denoted by CCA, on count array. It is of size $(u + 1)$. It is built as shown.

$$CCA[i] = \begin{cases} 0 & \text{if } i = 1, \\ CCA[i-1] + CA[i-1] & \text{if } i \in [2, u+1]. \end{cases}$$

Algorithm COMBINATIONS incorporates the observations described above and its running time is $O((n - r + 1) \times r \times |\mathcal{C}^r(L)|)$, where $|\mathcal{C}^r(L)|$ denotes the number of r-combinations of L.

[1] In the $(r + 1)^{th}$ level, the permutations are output.

```
PERMUTATIONS($index$)
if $index > r$ then print $\mathcal{R}[1..r]$
else
  for $i \leftarrow 1$ to $u$ do
    if $CA[i] \geq 1$ then
      $\mathcal{R}[index] \leftarrow U[i]$
      $CA[i] \leftarrow CA[i] - 1$
      PERMUTATIONS($index + 1$)
      $CA[i] \leftarrow CA[i] + 1$
```

```
SUBSETS($index$)
print $\mathcal{R}[1..index]$
if $index > n$ then return
else
  for $i \leftarrow 1$ to $u$ do
    if $CA[i] \geq 1$ and $(index = 1$
    or $i > P[\mathcal{R}[index - 1]])$ then
      $\mathcal{R}[index] \leftarrow U[i]$
      $CA[i] \leftarrow CA[i] - 1$
      SUBSETS($index + 1$)
      $CA[i] \leftarrow CA[i] + 1$
```

```
CATALAN($index$)
if $index > 2n$ then print $\mathcal{R}[1..2n]$
else
  for $i \leftarrow 1$ to $2$ do
    if $CA[i] \geq 1$ and $(i \neq 1$
    or $CA[2] > CA[1])$ then
      $\mathcal{R}[index] \leftarrow U[i]$
      $CA[i] \leftarrow CA[i] - 1$
      CATALAN($index + 1$)
      $CA[i] \leftarrow CA[i] + 1$
```

```
COMPOSITIONS($index$)
if $n = 0$ then print $\mathcal{R}[1..(index - 1)]$
else
  for $i \leftarrow 1$ to $u$ do
    if $n \geq U[i]$ then
      $\mathcal{R}[index] \leftarrow U[i]$
      $n \leftarrow n - U[i]$
      COMPOSITIONS($index + 1$)
      $n \leftarrow n + U[i]$
```

```
PALINDROMES($index$)
if $index = \lfloor r/2 \rfloor$ then
  if $r\%2 = 1$ then
    for $i = 1$ to $u$ do
      if $CA[i] \geq 1$ then
        $\mathcal{R}[index] \leftarrow U[i]$
        print $\mathcal{R}[1..r]$
  else print $\mathcal{R}[1..r]$
else
  for $i \leftarrow 1$ to $u$ do
    if $CA[i] \geq 2$ then
      $\mathcal{R}[index] \leftarrow \mathcal{R}[r - index + 1] \leftarrow U[i]$
      $CA[i] \leftarrow CA[i] - 2$
      PALINDROMES($index + 1$)
      $CA[i] \leftarrow CA[i] + 2$
```

```
SUDOKU($row, col$)
if $col > n$ then $row \leftarrow row + 1; col = 1$
if $row > n$ then print $\mathcal{R}[1..n][1..n]$
elseif $R[row][col] > 0$ then
  SUDOKU($row, col + 1$)
else
  for $i \leftarrow 1$ to $n$ do
    $block \leftarrow (row - 1) \times n + col$
    if $CA^{row}[row][i] \geq 1$ & $CA^{col}[col][i] \geq 1$ &
      $CA^{block}[block][i] \geq 1$ then
      $\mathcal{R}[row][col] \leftarrow i$
      $CA^{row}[row][i] - -; CA^{col}[col][i] - -$
      $CA^{block}[block][i] - -$
      SUDOKU($row, col + 1$)
      $CA^{row}[row][i] + +; CA^{col}[col][i] + +$
      $CA^{block}[block][i] + +$
```

```
COMBINATIONS($index$)
if $index > r$ then print $\mathcal{R}[1..r]$
else
  $i \leftarrow P[\mathcal{R}[index - 1]]$
  while $(CCA[u + 1] - CCA[i + 1]$
  $+ CA[i]) > (r - index)$ do
    if $CA[i] \geq 1$ then
      $\mathcal{R}[index] \leftarrow U[i]$
      $CA[i] \leftarrow CA[i] - 1$
      COMBINATIONS($index + 1$)
      $CA[i] \leftarrow CA[i] + 1$
    $i \leftarrow i + 1$
```

```
DIOPHANTINE($index$)
if $index = u$ & $sum \neq 0$ then return
elseif $sum = 0$ print $\mathcal{R}[1..u]$
else
  for $i \leftarrow 0$ to $CA[index]$ do
    if $sum \geq i \times U[index]$ then
      $\mathcal{R}[index] \leftarrow i$
      $CA[i] \leftarrow CA[i] - i$
      $sum \leftarrow sum - i \times U[index]$
      DIOPHANTINE($index + 1$)
      $CA[i] \leftarrow CA[i] + i$
      $sum \leftarrow sum + i \times U[index]$
```

```
PARTITIONS($index$)
if $n = 0$ then print $\mathcal{R}[1..(index - 1)]$
else
  $i \leftarrow P[\mathcal{R}[index - 1]]$
  while $(CU[u + 1] - CU[i + 1]$
  $+ U[i]) > n$ do
    if $n \geq U[i]$ then
      $\mathcal{R}[index] \leftarrow U[i]$
      $n \leftarrow n - U[i]$
      PARTITIONS($index + 1$)
      $n \leftarrow n + U[i]$
```

```
GRAY-CODE($index$)
if $index > n$ then
  print $\mathcal{R}[1..(index - 1)]$
else
  for $i \leftarrow 0$ to $1$ do
    if $i = 0$ then $\mathcal{R}[index] \leftarrow CA[index]$
    else $\mathcal{R}[index] \leftarrow 1 - CA[index]$
    $CA[index + 1] \leftarrow CA[index + 1] + i$
    GRAY-CODE($index + 1$)
    $CA[index + 1] \leftarrow CA[index + 1] - i$
```

```
DERANGEMENTS($index$)
if $index > r$ then print $\mathcal{R}[1..r]$
else
  for $i \leftarrow 1$ to $u$ do
    if $CA[i] \geq 1$ & $L[index] \neq U[i]$ then
      $\mathcal{R}[index] \leftarrow U[i]$
      $CA[i] \leftarrow CA[i] - 1$
      DERANGEMENTS($index + 1$)
      $CA[i] \leftarrow CA[i] + 1$
```

```
N-QUEENS($index$)
if $index > n$ then
  print $\mathcal{R}[1..(index - 1)]$
else
  for $i \leftarrow 1$ to $n$ do
    if $CA^{col}[i] \geq 1$ &
      $CA^{ld}[n + index - i] \geq 1$ &
      $CA^{rd}[index + i - 1] \geq 1$ then
      $\mathcal{R}[index] \leftarrow i$
      $CA^{col}[i] - -$
      $CA^{ld}[n + index - i] - -$
      $CA^{rd}[index + i - 1] - -$
      N-QUEENS($index + 1$)
      $CA^{col}[i] + +$
      $CA^{ld}[n + index - i] + +$
      $CA^{rd}[index + i - 1] + +$
```

Figure 2 Combinatorial algorithms.

Subsets. True to the formula $2^n = \sum_{r=0}^{n} \binom{n}{r}$, we know that all subsets of a list can be generated using an algorithm that generates all r-combinations. This is achieved by calling the r-combinations generating algorithm successively with: $0 \leq r \leq n$. However, the framework structure allows us to do it more efficiently.

All subsets L can be obtained by calling SUBSETS algorithm. It has the additional conditional check ($index = 1$ **or** $i \geq P[\mathcal{R}[index - 1]]$) before assigning an element to $\mathcal{R}$. We add a **print** statement before the **if** block, where r is replaced by n. This makes sure that we print a subset as and when they are generated in lexicographic order. The count array is built exactly as explained in PERMUTATIONS algorithm.

Catalan family. Our framework can be used to output all possible valid parenthesizations of $(n+1)$ terms. A valid parenthesization of $(n+1)$ terms can be defined as a sequence of n opening brackets and n closing brackets with the condition that at no point in the sequence should the number of closing brackets be greater than the number of opening brackets. We shall refer to the set of all possible valid parenthesizations of $(n+1)$ terms as n-parenthesizations.

We initialize CA such that $CA[1] = CA[2] = n$. This can be viewed as an input list of n opening brackets and n closing brackets ($u = 2$). We then generate all possible permutations of this input list using CA and make sure that at no point in a permutation we assign more closing brackets than opening brackets. We set $U[1] =$ "(" and $U[2] =$ ")". We invoke CATALAN to generate all valid parenthesizations.

Integer compositions & partitions. A composition of a natural number n is an arrangement of strictly positive integers which sum up to n. For example, 3 has four compositions - $\{\langle 1,1,1\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 3\rangle\}$. The total number of compositions of n is $\sum_{r=1}^{n-1} \binom{n-1}{r} = 2^{n-1}$. To generate the compositions of n we set $u = n$, replace $U[i]$ by i in the algorithm COMPOSITIONS.

It is important to note that the pseudocode COMPOSITIONS is doing much more than generating standard compositions. Generally, a composition of n can contain any number from 1 to n. However, the algorithm COMPOSITIONS, based on the same algorithmic structure used so far, that given an n and $U[1..u]$, generates all possible compositions of n using elements in U. It is also possible to generate compositions when we have a limited number of some or all of the numbers. For example, we could have $n = 15$; $U = \langle 2, 3\rangle$ with the condition that 2 and 3 cannot be used more than 6 and 3 times respectively. This can be handled by maintaining count array $CA = \langle 6, 3\rangle$ parallel to U.

A partition of a natural number n is a set (order does not matter) of strictly positive integers which sum up to n. For example, 3 has three partitions - $\{\langle 1,1,1\rangle, \langle 1,2\rangle, \langle 3\rangle\}$. The number of partitions of n is the coefficient of x^n in the expansion of $\Pi_{j=1}^{\infty}(1/(1-x^j))$. Analogous to how we constructed COMBINATIONS from PERMUTATIONS, we can construct PARTITIONS from COMPOSITIONS. We define cumulative unique array CU as below and use it in the partition algorithm.

$$CU[i] = \begin{cases} 0 & \text{if } i = 1, \\ CU[i-1] + U[i-1] & \text{if } i \in [2, u+1]. \end{cases}$$

Diophantine equation with positive coefficients. A Diophantine equation is a polynomial equation in which only integer solutions are allowed. In this section we consider linear Diophantine equations that have the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$ where, a_i, b are natural numbers for all $i \in [1, n]$, and only non-negative solutions are allowed.

The algorithm DIOPHANTINE gives a procedure to generate all solutions. We set $sum = b$, $u = n$, $U[i] = a_i$, $CA[i] = \lfloor b/a_i \rfloor$, and invoke DIOPHANTINE. Given that the coefficients are always positive and only non-negative solutions are allowed, then to assign a particular value for $\mathcal{R}[index]$ (or x_{index}), we loop in the range $[0, b/a_{index}]$ decrementing and incrementing the sum appropriately. It is also possible to optimize the algorithm as shown in Combinations.

We can also extend the algorithm to generate all non-negative solutions to the class of Diophantine equation of the form $\sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}x_i^{m-j+1} = b$, where, $a_{ij}, b > 0$ and are constants for all $i \in [1, n]$.

Gray code. Gray code (aka reflected binary code) is a binary system in which two consecutive binary codes differ in only a single bit. Gray codes are not unique. For example, gray codes for two bits can be $\langle 00, 01, 11, 10\rangle$ or $\langle 00, 10, 11, 01\rangle$.

Given the number of bits, n, the Gray-Code algorithm generates 2^n gray codes by constructing a recursion tree with branch factor 2 and height n. The array $CA[index+1]$ holds information whether the visited node at level $index+1$ is the left or right child of its parent node (which is at level $index$). If a node at level $index+1$ is the left child of its parent then we set $CA[index+1] = 0$, else we set $CA[index+1] = 1$. At every node, we loop through 0 and 1, meaning scanning both left and right branches of the node and setting $\mathcal{R}[index]$ with $CA[index]$ if it is the left child of its parent or $1 - CA[index]$ if it is the right child. The recursion tree of Gray-Code for three bits is given in Figure 3.

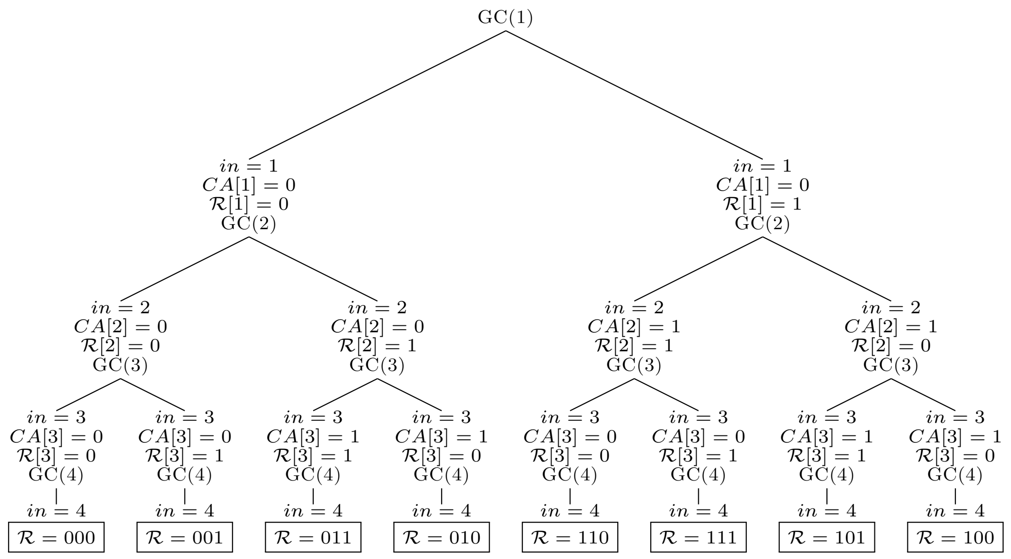
GC(1)
in = 1, CA[1] = 0, R[1] = 0, GC(2)
in = 1, CA[1] = 0, R[1] = 1, GC(2)
in = 2, CA[2] = 0, R[2] = 0, GC(3)
in = 2, CA[2] = 0, R[2] = 1, GC(3)
in = 2, CA[2] = 1, R[2] = 1, GC(3)
in = 2, CA[2] = 1, R[2] = 0, GC(3)
in = 3, CA[3] = 0, R[3] = 0, GC(4)
in = 3, CA[3] = 0, R[3] = 1, GC(4)
in = 3, CA[3] = 1, R[3] = 1, GC(4)
in = 3, CA[3] = 1, R[3] = 0, GC(4)
in = 3, CA[3] = 0, R[3] = 0, GC(4)
in = 3, CA[3] = 0, R[3] = 1, GC(4)
in = 3, CA[3] = 1, R[3] = 1, GC(4)
in = 3, CA[3] = 1, R[3] = 0, GC(4)
in = 4
R = 000
R = 001
R = 011
R = 010
R = 110
R = 111
R = 101
R = 100

Figure 3 Recursion tree of Gray-Code for $n = 3$. The names Gray-Code and $index$ are denoted by GC and in, respectively.

▶ **Theorem 1.** *The algorithms given in Figure 2 are correct.*

Due to space constraints, the correctness proofs of the algorithms are given in Section B of Appendix. Theorems 2 and 3 in Appendix proves the correctness of the Permutations and Gray-Code algorithms, respectively. Other algorithms derive naturally from the Permutations algorithm and hence proving them is straightforward.

Palindromes. A palindrome is a sequence of characters that reads the same both forwards and backwards. E.g.: Malayalam. To generate palindromes given a multiset of characters, we construct CA and U in a way similar as shown in the permutations algorithm and then invoke Palindromes. Before assigning any character for an index in $\mathcal{R}$, we loop through all possible unique characters and check if two instances of the same character are available. If available, then we assign that character to both $\mathcal{R}[index]$ and $\mathcal{R}[r-index+1]$ and decrement the character's count by two. We continue the process until the index is half the size of r. At this point, if we need even-sized palindrome we simply print the palindromes or if we need odd-sized palindrome we substitute any character as the middle character of the palindrome and print it.

Derangements. An r-derangement of a list is an r-permutation of the list in which none of the elements appear in their original positions. With one additional check, PERMUTATIONS algorithm can be transformed into DERANGEMENTS. Before assigning an element at a particular index, we perform a check to ensure that the element $U[i]$ does not occur at the exact same index in L, i.e., if we were to assign $U[i]$ to $\mathcal{R}[index]$, we would need to ensure that $L[index] \neq U[i]$. The array CA is built exactly as shown in PERMUTATIONS.

Sudoku. Sudoku is a number puzzle where a partially filled 9×9 grid must be completed filled with numbers from 1 to 9 such that each row, each column, and the nine 3×3 blocks contains all digits from 1 to 9. We generate all solutions to a partially filled $n \times n$ sudoku from the algorithm SUDOKU. The $R[1..n][1..n]$ represents the sudoku matrix and some cells in the matrix are already filled. The unfilled cells are initialized to 0. We define three count arrays: CA^{row}, CA^{col}, and CA^{block} corresponding to the information about which numbers have been used in different rows, columns, and blocks, respectively. If $R[r][c] = k \in [1, n]$, then $CA^{row}[r][k], CA^{col}[c][k]$, and $CA^{block}[b][k]$ are set to 0, where b is the block number found as $b = (r-1)n + c$. The remaining terms are set to 1.

For any unfilled cell (row, col) we loop through all numbers from $i \leftarrow 1$ to n and check whether each of the numbers i is feasible by checking whether that number i has appeared in anywhere in the row row or column col or block $((row-1)n + col)$ by simply checking the corresponding count arrays. If all of the CA values are 1, it means that the number i if feasible and we set $R[row][col]$ with that number and decrement the corresponding count array values. This process continues until all solutions to the sudoku are found.

***n*-Queens.** n-queens puzzle is a chessboard puzzle where n queens have to be placed on an $n \times n$ chessboard such that no two queens attack each other. This means that no two queens should be placed on the same row, or on the same column, or on the same diagonal (left or right). The algorithm N-QUEENS derived from the framework again matches with that given by Rohl [38].

The term $\mathcal{R}[i] = j$ means that the ith queen is placed at cell (i, j). Similar to sudoku we define three count arrays CA^{col}, CA^{ld}, and CA^{rd} corresponding to the information about which locations have been used in different columns, left (\) and right (/) diagonals, respectively. Initially we set $CA^{col}[1..n] = \{1, \ldots, 1\}$ and $CA^{ld}[1..2n-1] = CA^{rd}[1..2n-1] = \{1, \ldots, 1\}$. For every index, we loop through values from 1 to n and check if all the three count arrays are non-zero. If they are, then we set the value, decrement and increment the count array values appropriately to generate all solutions to the problem.

4 Related Work

The field of combinatorial algorithms is too vast to cover in a single paper or even in a single book. – Robert E. Tarjan.

For a nice introduction to combinatorial algorithms, please refer the book by Brualdi [8]. Excellent coverage on the topic is given by Kreher & Stinson [25], Knuth [22], Ruskey [39], Nijenhuis & Wilf [34], and Arndt [5].

Permutations, combinations, & subsets. Algorithms to generate all $n!$ permutations of a list of n elements are given by Wells [47], Heap [19], Ives [20], Fike [14], Lipski [28], Johnson [21], Trotter [46], Zaks [48], Rohl [38], and others. The permutation algorithms can use any of the following approaches: swaps, adjacent swaps, reversals of prefix, counters for elements, rotations of prefix, ranking / unranking, and additions in base-n. For thorough treatment on different permutation algorithms, please refer to the surveys by Ord-Smith [36][37], Sedgewick [41], and Knuth [22]. The limitation most permutation algorithms have is that they do not have the capability to efficiently generate all unique r-permutations for a multiset.

An unordered selection of a list of elements is called a combination. Algorithms to generate combinations are given by Mifsud [30], Shen [42], Chase [10], [13], Kurtzberg [26], Bitner et al. [6], Liu & Tang [29], Knuth [22], Eades & McKay [12], Akl et al. [4], and Akl [3]. Most of such algorithms generate r-combinations. Few algorithms work with multisets. Surveys on combination algorithms are given by Akl [2] and Carkeet & Eades [9]. Some applications of combinations are given by Stojmenović [43].

The terms r-combinations and r-subsets mean the same. Therefore, the algorithms that produce combinations can also be used to generate subsets. Subsets are computed in one of the two ways: (a) generating r-subsets for all $r \in [0, n]$; and (b) generating all subsets where the subsets are no longer sorted by their sizes. To make subset generation easy, the 2^n subsets of an n-element set are often represented by n-length bit strings, where a 1-bit denotes that the corresponding element is present, and 0-bit denotes its absence. This means that any systematic way to generate all such bit strings also produces subsets. Nevertheless, it is still difficult to generate subsets for a multiset.

Compositions, partitions, & Catalan families. Algorithms for compositions are given by Ehrlich [13], Ruskey [39], and Stojmenović [44]. Algorithms to generate partitions are given by Knuth [22], Stojmenović [44], and Stojmenović & Zoghbi [45]. Algorithm to generate restricted versions of compositions and partitions is given by Opdyke [35].

A beautiful essay on Catalan numbers is written by Gardner [15]. Koshy's book [24] gives several applications of Catalan numbers. Algorithms to generate binary trees are given by Knuth [22].

Diophantine equations, gray code, & derangements. General algorithms to find solutions to a linear Diophantine equation were given by Bond [7], Mrit & Salkin [32], and Chou & Collins [11].

Algorithms to generate binary reflected gray codes are given by Gray [18], Gilbert [16], and Bitner, Ehrlich, & Reingold [6]. Combinatorial gray codes and their uses are surveyed by Savage [40], Knuth [22] and Ruskey [39].

Derangements-generating algorithms are given by Akl [1], Korsh & LaFollette [23], and Mikawa & Semba [31].

References

1 Selim G Akl. A new algorithm for generating derangements. *BIT Numerical Mathematics*, pages 20(1):2–7, 1980.

2 Selim G Akl. A comparison of combination generation methods. *TOMS*, pages 7(1):42–45, 1981.

3 Selim G. Akl. Adaptive and optimal parallel algorithms for enumerating permutations and combinations. *The Computer Journal*, pages 30(5):433–436, 1987.

4 Selim G Akl, David Gries, and Ivan Stojmenovic. An optimal parallel algorithm for generating combinations. *IPL*, pages 33(3):135–139, 1989.

5 Jorg Arndt. *Matters Computational: Ideas, Algorithms, Source Code.* Springer-Verlag New York, Inc., 2010.

6 James R Bitner, Gideon Ehrlich, and Edward M Reingold. Efficient generation of the binary reflected gray code and its applications. *CACM*, pages 19(9):517–521, 1976.

7 James Bond. Calculating the general solution of a linear diophantine equation. *American Mathematical Monthly*, pages 955–957, 1967.

8 Richard A Brualdi. *Introductory combinatorics.* 2009.

9 Margaret Carkeet and Peter Eades. Performance of subset generating algorithms. *Annals of Discrete Math*, pages 26:49–58, 1985.

10 Phillip J Chase. Algorithm 382: combinations of m out of n objects [g6]. *CACM*, page 13(6):368, 1970.

11 Tsu-Wu J Chou and George E Collins. Algorithms for the solution of systems of linear diophantine equations. *SIAM Journal on computing*, pages 11(4):687–708, 1982.

12 Peter Eades and Brendan McKay. An algorithm for generating subsets of fixed size with a strong minimal change property. *IPL*, pages 19(3):131–133, 1984.

13 Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *JACM*, pages 20(3):500–513, 1973.

14 CT Fike. A permutation generation method. *The Computer Journal*, pages 18(1):21–22, 1975.

15 Martin Gardner. *Time travel and other mathematical bewilderments.* 1988.

16 Edgard N Gilbert. Gray codes and paths on the n-cube. *Bell System Technical Journal*, pages 37(3):815–826, 1958.

17 Martin Charles Golumbic and Irith Ben-Arroyo Hartman. *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications.* Springer, 2005.

18 Frank Gray. Pulse code communication, March 17 1953. US Patent 2,632,058.

19 BR Heap. Permutations by interchanges. *The Computer Journal*, pages 6(3):293–298, 1963.

20 FM Ives. Permutation enumeration: four new permutation algorithms. *CACM*, pages 19(2):68–72, 1976.

21 Selmer M Johnson. Generation of permutations by adjacent transposition. *Mathematics of computation*, pages 17(83):282–285, 1963.

22 Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms: Part 1.* 2011.

23 James F Korsh and Paul S LaFollette. Constant time generation of derangements. *IPL*, pages 90(4):181–186, 2004.

24 Thomas Koshy. Catalan numbers with applications. 2008.

25 Donald L Kreher and Douglas R Stinson. *Combinatorial algorithms: generation, enumeration, and search*, volume 7. CRC press, 1998.

26 Jerome Kurtzberg. Algorithm 94: Combination. *CACM*, page 5(6):344, 1962.

27 Anany V Levitin. *Introduction to the Design & Analysis of Algorithms.* Pearson, 2011.

28 W Lipski Jr. More on permutation generation methods. *Computing*, pages 23(4):357–365, 1979.

29 CN Liu and DT Tang. Algorithm 452: enumerating combinations of m out of n objects [g6]. *CACM*, page 16(8):485, 1973.

30 Charles J Mifsud. Algorithm 154: combination in lexicographical order. *CACM*, page 6(3):103, 1963.

31 Kenji Mikawa and Ichiro Semba. Generating derangements by interchanging at most four elements. *Systems and Computers in Japan*, pages 35(12):25–31, 2004.

32 Susumu Mrit and Harvey M Salkin. Finding the general solution of a linear diophantine equation. 1979.

33 Amiya Nayak and Ivan Stojmenovic. *Handbook of applied algorithms: Solving scientific, engineering, and practical problems.* John Wiley & Sons, 2007.

34 Albert Nijenhuis and Herbert S Wilf. *Combinatorial algorithms: for computers and calculators.* Elsevier, 2014.

35 John Douglas Opdyke. A unified approach to algorithms generating unrestricted and restricted integer compositions and integer partitions. *Journal of Mathematical Modelling and Algorithms*, pages 9(1):53–97, 2010.

36 RJ Ord-Smith. Generation of permutation sequences: part 1. *The Computer Journal*, pages 13(2):152–155, 1970.

37 RJ Ord-Smith. Generation of permutation sequences: Part 2. *The Computer Journal*, pages 14(2):136–139, 1971.

38 Jeffrey S. Rohl. Generating permutations by choosing. *The Computer Journal*, pages 21(4):302–305, 1978.

39 Frank Ruskey. Combinatorial generation. 2001.

40 Carla Savage. A survey of combinatorial gray codes. *SIAM review*, pages 39(4):605–629, 1997.

41 Robert Sedgewick. Permutation generation methods. *ACM Computing Surveys (CSUR)*, pages 9(2):137–164, 1977.

42 Mok-kong Shen. On the generation of permutations and combinations. *BIT Numerical Mathematics*, pages 2(4):228–231, 1962.

43 I Stojmenović and Masahiro Miyakawa. Applications of a subset-generating algorithm to base enumeration, knapsack and minimal covering problems. *The Computer Journal*, pages 31(1):65–70, 1988.

44 Ivan Stojmenovic. *Handbook of Applied Algorithms. Chapter: Generating all and random instances of a combinatorial object.* Wiley, 2008.

45 Ivan Stojmenović and Antoine Zoghbi. Fast algorithms for generating integer partitions. *International Journal of Computer Mathematics*, pages 70(2):319–332, 1998.

46 HF Trotter. Algorithm 115: Perm. *CACM*, pages 5(8):434–435, 1962.

47 Mark B Wells. Generation of permutations by transposition. *Mathematics of Computation*, pages 192–195, 1961.

48 Shmuel Zaks. A new algorithm for generation of permutations. *BIT Numerical Mathematics*, pages 24(2):196–204, 1984.

Appendix

A Combinatorial Algorithms

A.1 Permutations

We venture an example to better illustrate the process of generating permutations. Assume we have $L = [a, a, b, b]$ and $r = 3$. From L, we find that $n = 4, u = 2, U = [a, b]$, and $CA = [2, 2]$. These variables are input to Algorithm PERMUTATIONS. The recursion tree that would be obtained is shown in Figure 4. The root of the tree represents the initial state, i.e. $CA = [2, 2]$ and $\mathcal{R} = [\ ]$. Every descendant of the root represents a particular recursion depth, indicated by the value of *index*. At every node, we indicate the values in CA and $\mathcal{R}$ just before going down to the next recursion level followed by the function call that initiates the next recursion level. Once we go four levels deep into the recursion, i.e. $index = 4$, the contents of $\mathcal{R}$ would be output because $index > r$. The control then returns to the previous recursion level. The process continues until all permutations are generated.

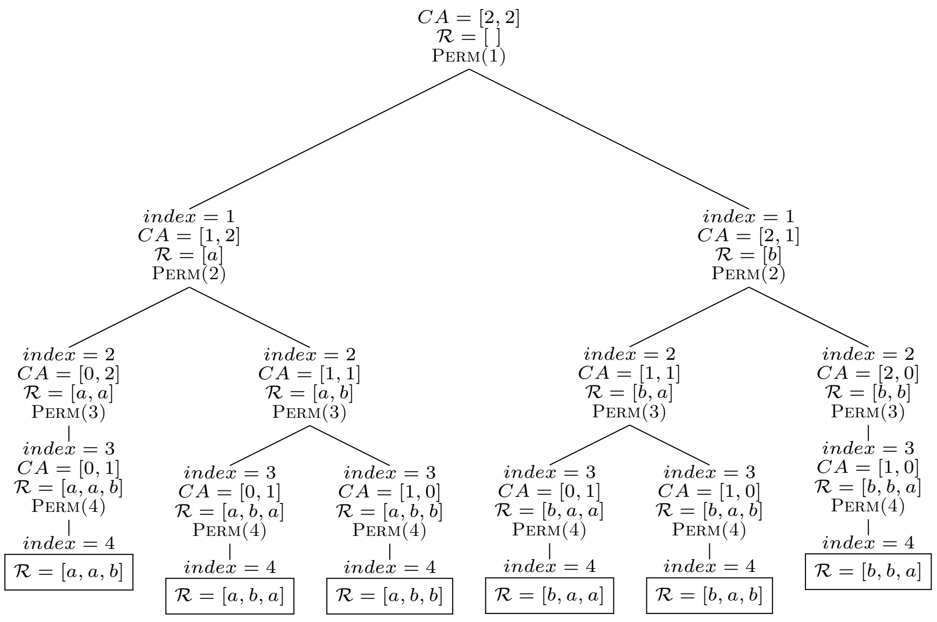

Figure 4 Recursion tree of PERMUTATIONS for $L = [a, a, b, b]$ and $r = 3$.

A.2 Combinations

Recursion tree for naive version of COMBINATIONS. One simple change to PERMUTATIONS algorithm can produce all possible r-combinations of an input list. Before assigning an element at a particular index in $\mathcal{R}$, we make sure that the element occupies a position in L that is greater than the position of element at the previous index. At the first index however, we may assign any element. The additional condition required is: $(index = 1$ **or** $i \geq P[\mathcal{R}[index - 1]])$. We use the condition $i > P[\mathcal{R}[index - 1]]$ to get combinations that contain no repeated elements.

The recursion tree obtained when we generate all combinations of $L = [a, b, c]$ with $r = 2$ is shown in Figure 5. As is evident from Figure 5, there can be many wasteful branches with certain kinds of input. For example, if we were to have $L = [a, b, \ldots, y, z]$ and $r = 26$, we would have a recursion tree similar to Figure 6, where branches ending with $\emptyset$ denote wasteful branches, i.e., recursion branches that do not produce any output. All unnecessary computations are because we loop from $1 \ldots u$ at every index.

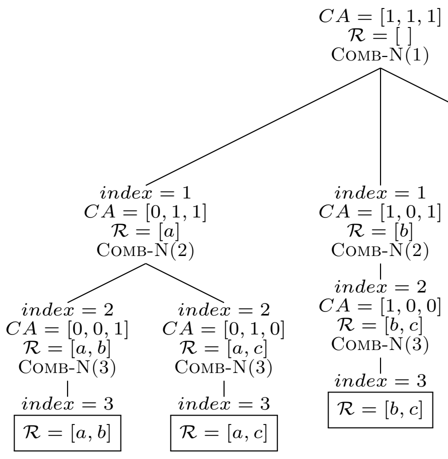

Figure 5 Recursion tree for the naive version of COMBINATIONS, denoted by COMB-N, for $L = [a, b, c]$ and $r = 2$.

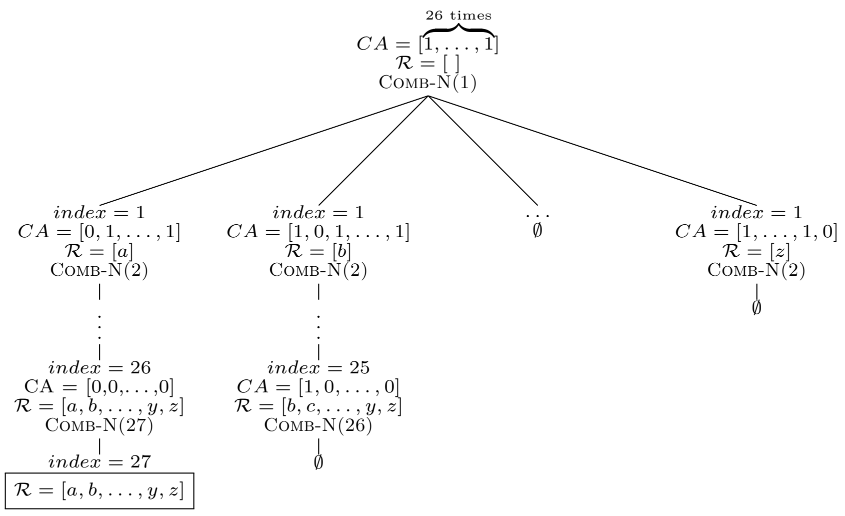

Figure 6 Recursion tree for the naive version of COMBINATIONS, denoted by COMB-N, for $L = [a, b, c, \ldots, z]$ and $r = 26$.

Recursion tree for COMBINATIONS. The algorithm is described in Figure 2. Figure 7 gives the recursion tree of COMBINATIONS for $L = [a, b, c, d]$ and $r = 3$.

A.3 Catalan Family

Figure 8 gives the recursion tree for CATALAN for $n = 3$. The Catalan number for $n = 3$ is 5. Hence, there are 5 leaves in the recursion tree. Each internal-node can have one or two child

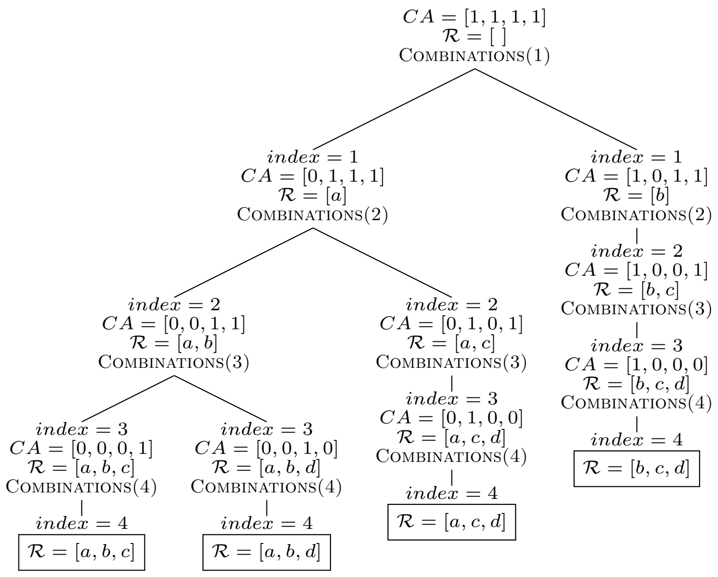

Figure 7 Recursion tree of COMBINATIONS for $L = [a, b, c, d]$ and $r = 3$.

nodes. The number of child nodes depends on how many times the condition $CA[2] > CA[1]$ is satisfied.

A.4 Integer Compositions

We define O to be the order list, where $O[i]$ is considered lexicographically earlier than $O[j]$ for $i \leq j$. Figure 9 gives the recursion tree of COMPOSITIONS for $n = 9$ and $O = [3, 2]$.

B Proofs of Correctness

▶ **Theorem 2** (PERMUTATIONS correctness). *The* PERMUTATIONS *algorithm generates all valid unique r-permutations of L.*

Proof. For simplicity, we use PERM to denote PERMUTATIONS, $\mathcal{P}_i$ to denote a general r-permutation instead of $\mathcal{P}_i^r$, and $\mathcal{P}_i[x]$ to denote the xth element of the ith permutation. To prove the theorem, we need to show the following three: (1) PERM generates valid permutations of L, (2) PERM generates unique permutations of L, and (3) PERM generates unique all r-permutations of L.

(1) PERM *generates valid permutations of L.*
We assign an element $U[i]$ to any particular index in $\mathcal{R}$ only if we have $CA[i] \geq 1$. Also, we decrement or increment $CA[i]$ whenever $U[i]$ is assigned or de-assigned, respectively. This process ensures that PERM only assigns elements that were actually present in L and no element is assigned more times than it occurs.

(2) PERM *generates unique permutations of L.*
The algorithm generates the next permutation from previous permutation as follows. Once a permutation is output, the program searches backwards from the end of $\mathcal{R}$ for an index where it can assign, from the available elements, an element that has a *higher* position in U than the previously assigned element. This means that, if $U[x]$ were assigned at a particular index, the algorithm checks if it can assign any among $U[x+1] \ldots U[u]$ at the

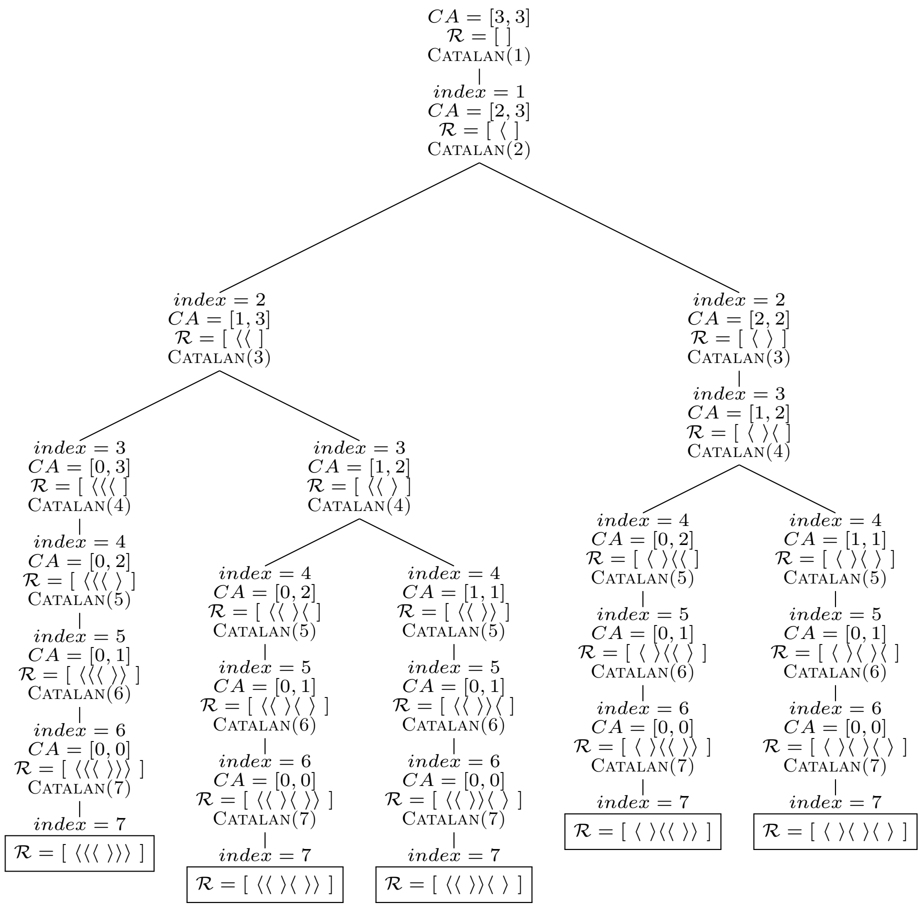

Figure 8 Recursion tree of Catalan for $n = 3$, i.e., $CA = [3, 3]$. For clarity we use angle brackets instead of parentheses.

same index. At the *first instance* (referred to as Property 1) of such an index, the algorithm assigns the *first among* (referred to as Property 2) $U[x + 1] \dots U[u]$, whichever is available. We define *discriminating index* d_{ij} between two different r-permutations $\mathcal{P}_i$ and $\mathcal{P}_j$ as the first (minimum) index in both permutations at which the elements differ from each other i.e., $d_{ij} = \min\{k : \mathcal{P}_i[k] \neq \mathcal{P}_j[k]\}$. Note that this index would be the *discriminating index* between the previously output permutation and the next permutation that is going to be output. Once this is done, the program *populates the rest of* $\mathcal{R}$ *minimally* (referred to as Property 3).

This process ensures that, between two consecutive permutations, we will have at least one index where the elements differ. Given that at this index, i.e., discriminating index, we assign one of $U[x + 1] \dots U[u]$, we can be sure that the permutation generated subsequent to a particular permutation will occupy a higher position in $\mathcal{P}^r(L)$.

(3) Perm *generates all* r*-permutations of* L.
Let d_{xy} denote the discriminating index between $\mathcal{P}_x$ and $\mathcal{P}_y$. Let $I(x)$ represent the index of the element x in U. Let three consecutive permutations generated by Perm be denoted by $\mathcal{P}_i$, $\mathcal{P}_j$, and $\mathcal{P}_k$. Also, let $\mathcal{P}_a$ and $\mathcal{P}_b$ be two consecutive permutations generated by Perm. Then, we prove the result in three parts.

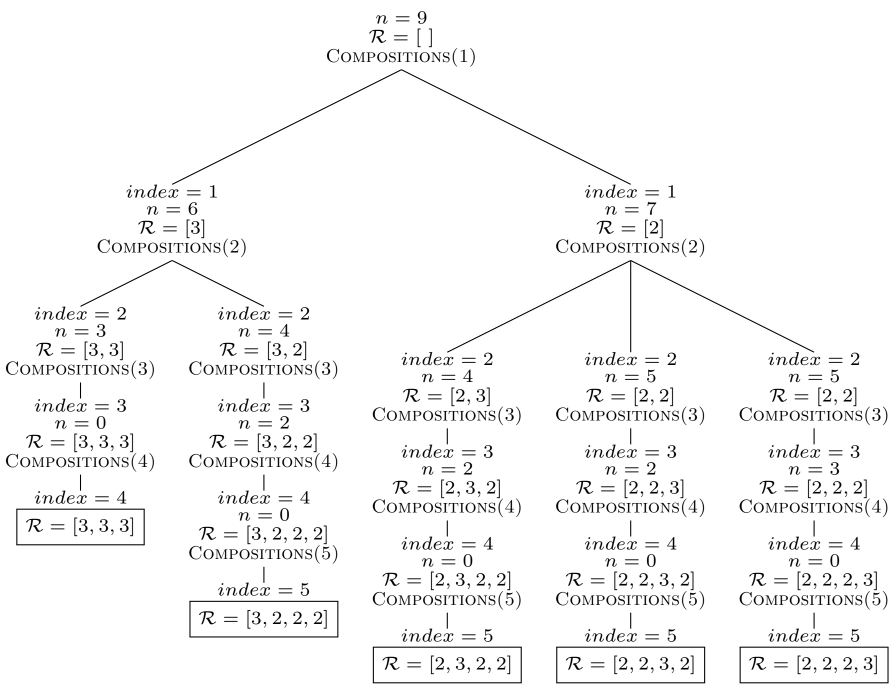

Figure 9 Recursion tree of COMPOSITIONS for $n = 9$ and $O = [3, 2]$. This means, as per the given example, 3 is lexicographically earlier than 2.

(i) $d_{ij} \geq d_{ik}$ *and* $d_{jk} \geq d_{ik}$.
We use contradiction. Let us assume we have $\mathcal{P}_i$, $\mathcal{P}_j$, $\mathcal{P}_j$ such that

$$d_{ij} < d_{ik} \tag{1}$$

By the definition of discriminating index, we have the following equations:

$$\mathcal{P}_i[1 \ldots (d_{ij} - 1)] = \mathcal{P}_j[1 \ldots (d_{ij} - 1)] \tag{2}$$
$$I(\mathcal{P}_i[d_{ij}]) < I(\mathcal{P}_j[d_{ij}]) \tag{3}$$
$$\mathcal{P}_i[1 \ldots (d_{ik} - 1)] = \mathcal{P}_k[1 \ldots (d_{ik} - 1)] \tag{4}$$
$$I(\mathcal{P}_i[d_{ik}]) < I(\mathcal{P}_k[d_{ik}]) \tag{5}$$

From equations 1 and 4, we can say $\mathcal{P}_k[d_{ij}] = \mathcal{P}_i[d_{ij}]$. By using this in equation 3, we get

$$I(\mathcal{P}_k[d_{ij}]) < I(\mathcal{P}_j[d_{ij}]) \tag{6}$$

Also, from equations 2 and 4 and equation 1, we get

$$\mathcal{P}_k[1 \ldots (d_{ij} - 1)] = \mathcal{P}_j[1 \ldots (d_{ij} - 1)] \tag{7}$$

Equations 6 and 7 imply that $\mathcal{P}_k$ comes before $\mathcal{P}_j$, when order of consideration is U, which is a contradiction to the order of the permutations. This in turn implies that the initial assumption is wrong. Hence, $d_{ij} \geq d_{ik}$.

By a similar argument, $d_{jk} \geq d_{ik}$.

(ii) $\nexists \mathcal{P}_c \in (\mathcal{P}^r(\mathcal{L})/\{\mathcal{P}_a, \mathcal{P}_b\})$*: such that* $\mathcal{P}_c$ *occurs between* $\mathcal{P}_a$ *and* $\mathcal{P}_b$.[2].
We use contradiction. Assume there exists such a $\mathcal{P}_c$. By the analysis of how PERM generates

[2] $\mathcal{A}/\{A_1, A_2\}$ denotes set $\mathcal{A}$ excluding elements A_1 and A_2

a permutation subsequent to an already generated one

$$d_{ac} \not> d_{ab} \qquad \text{[From Property 1]} \tag{8}$$

$$d_{cb} \not> d_{ab} \qquad \text{[From Property 3]} \tag{9}$$

Also we have $d_{ac} \not< d_{ab}$ and $d_{cb} \not< d_{ab}$ from result (i). Thus $d_{ac} = d_{bc} = d_{ab} (= \lambda \text{ say})$ is the only possibility.

For $\mathcal{P}_c$ to be between $\mathcal{P}_a$ and $\mathcal{P}_b$: $I(\mathcal{P}_a[\lambda]) < I(\mathcal{P}_c[\lambda]) < I(\mathcal{P}_b[\lambda])$. This is a direct contradiction to Property 2. Thus we can have no $\mathcal{P}_c$ in between $\mathcal{P}_a$ and $\mathcal{P}_b$.

(iii) Perm *generates all r-permutations of L.*
We begin by proving that Perm correctly generates the first permutation. By result (ii), we have proved that the process of generating the next permutation given the current one is accurate in that the immediate next permutation, in accordance with the order U, is generated. We then prove that Perm terminates once all permutations are generated.

At the beginning of the algorithm, we perform a minimal assignment at the first index, i.e., $index = 1$. The program then moves down a recursion level and once again performs a minimal assignment. This process continues until the last recursion level. In the $(r+1)$th recursion level, the program outputs the permutation. Clearly the first permutation that would be output by Perm would be formed by populating $\mathcal{R}[1 \dots r]$ minimally.

The program control returns from a level of recursion only after looping in $[1, u]$ is complete, i.e., when there are no more available elements that can be allotted at that corresponding index in $\mathcal{R}$. We know that the range $[1, u]$ is finite because the size of L is finite. Also the maximum recursion depth of the program is finite because r is finite. This would ensure that every recursion level would eventually end which in turn ensures that the program will eventually terminate. ◀

▶ **Theorem 3** (Gray-Code correctness). *The* Gray-Code *algorithm generates all valid gray codes of length n.*

Proof. The branch factor of each node in the recursion tree is 2. The program control returns to its parent node when $index > n + 1$. Hence, the number of levels in the recursion tree is n and by the virtue of it, the total number of leaf nodes in the tree will be 2^n. Once the algorithm completes traversing all 2^n leaves, it returns to its caller.

We need to prove that the binary codes of any two adjacent leaves in the recursion tree differs by exactly 1 bit. First, we will find the values taken by CA at any given recursion-level. Second, we will compute the values of $\mathcal{R}$ at any given recursion-level, using CA. Third, we will compute the binary codes output by the algorithm using $\mathcal{R}$. Finally we show that these binary codes are indeed the gray codes.

(1) [Computing CA.]
All count array values are initialized to 0. The count array values are assigned at line 7 of the algorithm. As there are two values of i i.e., 0 and 1, every internal node will have two children: left and right for $i = 0$ and $i = 1$ and Also, when the algorithm processes the parameter $index$, we assign $CA[index + 1]$ the value of i. We know that the number of nodes at recursion-level $index$ is 2^{index}. Then the value of $CA[index]$ for different nodes (from left to right) at recursion-level $index$ can be written as

$$CA[index] \text{ for nodes at recursion-level } index = \begin{cases} [0, 0] & \text{if } index = 1, \\ [0, 0, 1, 1]^{2^{index-2}} & \text{if } index > 1; \end{cases}$$

where $[a, \ldots, b]^c = [\ \underbrace{a, \ldots, b}_{\text{repeated } c \text{ times}}\ ]$.

(2) [Computing $\mathcal{R}$.]

From lines 5 and 6, we see that $\mathcal{R}[index]$ takes the value of $CA[index]$ if $i = 0$ and $\mathcal{R}[index]$ takes the value of $1 - CA[index]$ if $i = 1$. This implies, the value of $\mathcal{R}[index]$ for different nodes (from left to right) at recursion-level $index$ can be written as

$$\mathcal{R}[index] \text{ for nodes at recursion-level } index = \begin{cases} [0,1] & \text{if } index = 1, \\ [0,1,1,0]^{2^{index-2}} & \text{if } index > 1. \end{cases}$$

Thus, we have

$$\mathcal{R}[1] \text{ for nodes at recursion-level } 1 = [0,1]$$

$$\mathcal{R}[2] \text{ for nodes at recursion-level } 2 = [0,1,1,0]$$

$$\mathcal{R}[3] \text{ for nodes at recursion-level } 3 = [0,1,1,0,0,1,1,0]$$

$$\mathcal{R}[i] \text{ for nodes at recursion-level } i = [0,1,1,0]^{2^{i-2}} \qquad (\text{for } i \in [2,n])$$

(3) [Computing G.]

From the status / configuration of the recursion-tree described above, we need to prove that $R[1, \ldots, n]$ of any two adjacent leaves in the recursion-tree differ by exactly 1 bit. Let $G[i][j]$, where $i \in [1, 2^n]$ and $j \in [1, n]$ denote the jth bit of the ith binary code generated from the algorithm. We need to prove that $G[i]$'s are indeed the gray codes.

Consider the first bit of all gray codes i.e., $G[1, \ldots, 2^n][1]$. Using $\mathcal{R}[1]$ and using the knowledge that every node at level 1 will have 2^{n-1} leaves in its subtree, we can write

$$G[1, \ldots, 2^n][1] = [\ \underbrace{0, \ldots, 0}_{2^{n-1} \text{ times}},\ \underbrace{1, \ldots, 1}_{2^{n-1} \text{ times}}\ ]$$

Similarly, using $\mathcal{R}[2]$ and using the knowledge that every node at level 2 will have 2^{n-2} leaves in its subtree, we can write

$$G[1, \ldots, 2^n][2] = [\ \underbrace{0, \ldots, 0}_{2^{n-2} \text{ times}},\ \underbrace{1, \ldots, 1}_{2^{n-2} \text{ times}},\ \underbrace{1, \ldots, 1}_{2^{n-2} \text{ times}},\ \underbrace{0, \ldots, 0}_{2^{n-2} \text{ times}}\ ]$$

Continuing the process till recursion-level n, we have

$$G[1, \ldots, 2^n][n] = [0,1,1,0]^{2^{n-2}}$$

It is easy to see that $G[1, \ldots, 2^n][1, \ldots, n]$ is exactly a gray code pattern. Hence, the Gray-Code algorithm is correct. ◀