Ancestral Colorings of Perfect Binary Trees
With Applications in Private Retrieval of Merkle Proofs

Quang Cao¹, Rinaldo Gagiano¹, Duy Huynh¹, Xun Yi¹, Son Hoang Dau¹,
Phuc Lu Le², Quang-Hung Luu³, Emanuele Viterbo⁴, Yu-Chih Huang⁵,
Jingge Zhu⁶, Mohammad M. Jalalzai⁷, and Chen Feng⁷

Abstract

We introduce a novel tree coloring problem in which each node of a rooted tree of height \( h \) is assigned one of the \( h \) colors under the condition that any two nodes that are ancestor and descendant of each other must have different colors and moreover, the numbers of nodes in any two distinct color classes differ by at most one. We refer to such a coloring as a balanced ancestral coloring. Our key contribution is to characterize, based on majorizations, all color sequences (not only the balanced ones) for which there exists an ancestral coloring for perfect binary trees. We then develop an almost linear-time divide-and-conquer algorithm to generate such a coloring for every perfect binary tree of height \( h \geq 1 \). The existence of a balanced ancestral coloring reveals an interesting fact about combinatorial batch code: when the batch follows a special pattern (consisting of nodes along a root-to-leaf path in a tree), the total storage capacity required can be reduced by a factor of \( \Theta(h) \) compared to when the batch is arbitrary while keeping a balanced storage capacity across \( h \) servers. Furthermore, our result also identifies an infinite family of graphs for which the equitable chromatic number can be explicitly determined. As far as we know, this family has not been discovered before in the literature. As a practical application, we show that a balanced ancestral coloring can be employed to speed up the private retrieval of a Merkle proof in a perfect binary tree by a factor of \( \Theta(h/2) \) compared to a straightforward parallel implementation of SealPIR, a state-of-the-art private information retrieval scheme.

I. Introduction

A. Problem Formulation

Motivated by the problem of private retrieval of Merkle proofs for Simplified Payment Verification [1] in blockchains, and more generally, in any verifiable data structure using Merkle trees (e.g., Google's Certificate Transparency [2], Google's Verifiable Data Structures and its implementation - Google’s Trillian [4], DataStax/Apache’s Cassandra [5], and Amazon’s DynamoDB [6]), we introduce and develop an efficient algorithm to solve a novel tree coloring problem defined as follows. Given a rooted tree of height \( h \), we aim to find a coloring of all tree nodes (excluding the root) using \( h \) colors 1, 2, \ldots, \( h \) satisfying the following two properties:

- (Ancestral Property) Each color class \( C_i \), which consists of tree nodes with color \( i \), doesn’t contain any pair of nodes \( u \) and \( v \) so that \( u \) is an ancestor of \( v \) (i.e., \( u \) lies in the path from \( v \) to the root and \( u \neq v \)), and
- (Balanced Property) \( |C_i| - |C_j| \leq 1 \), for every \( 1 \leq i, j \leq h \).

A coloring \( C = \{C_1, C_2, \ldots, C_h\} \) of a tree that satisfies the Ancestral Property is called an ancestral coloring. If \( C \) also satisfies the Balanced Property then it is called a balanced ancestral coloring. A trivial ancestral coloring is the coloring by layers, i.e., for a perfect binary tree, \( C_1 = \{2,3\}, C_2 = \{4,5,6,7\} \), etc. However, this layer-based coloring is not balanced. See Fig. 1 for examples.

¹Quang Cao, Rinaldo Gagiano, Duy Huynh, Xun Yi, and Son Hoang Dau are with the School of Computing Technologies, RMIT University, emails: {nhat.quang.cao, S3870806, duy.huynh}@student.rmit.edu.au and {xun.yi, son.hoang.dau}@rmit.edu.au. They are also with the RMIT University Centre for Cyber Security Research and Innovation (CCSRI). ²Phuc Lu Le is with University of Science, Vietnam National University, Ho Chi Minh City, email: lephucelu@gmail.com. ³Quang-Hung Luu is with Department of Computing Technologies, Swinburne University of Technology, email: hluu@swin.edu.au. ⁴Emanuele Viterbo is with Department of Electrical & Computer Systems Engineering, Monash University, email: emanuele.viterbo@monash.edu. ⁵Yu-Chih Huang is with Institute of Communications Engineering, National Chiao Tung University, email: jerryhuang@nctu.edu.tw. ⁶Jingge Zhu is with Department of Electrical and Electronic Engineering, the University of Melbourne, email: jingge.zhu@unimelb.edu.au. ⁷Mohammad M. Jalalzai and Chen Feng are with School of Engineering, the University of British Columbia (Okanagan Campus), email: {m.jalalzai, chen.feng}@ubc.ca.
of balanced ancestral colorings for the perfect binary trees $T(h)$ when $h = 1, 2, 3$. Note that these colorings may not be unique and different balanced ancestral colorings for $T(h)$ can be constructed when $h \geq 3$. Constructing balanced ancestral colorings (if any), or more generally, ancestral colorings with given color class sizes $c_i = |C_i|$ ($1 \leq i \leq h$), for every rooted tree, is a highly nontrivial task. As the first step toward this problem, we focus on perfect binary trees in this work.

Fig. 1. Balanced ancestral colorings for perfect binary trees $T(h)$ with $h = 1, 2, 3$. Nodes of the same color are not ancestors or descendants of each other and moreover, the color classes have sizes different from each other by at most one.

B. Color-Splitting Algorithm

In this paper, we develop a divide-and-conquer algorithm called Color-Splitting Algorithm (CSA) that takes $h$ as input and generates a balanced ancestral coloring for the perfect binary tree $T(h)$ of height $h$ in time almost linear in the number of tree nodes (more specifically, the running time is in $O(2^{h+1} \log h)$). In fact, this algorithm can generate not only a balanced ancestral coloring but also any ancestral coloring (color classes having heterogeneous sizes) feasible for $T(h)$ (see Definition 2 and Theorem 1). It is worth noting that this flexibility of our algorithm establishes the existence of optimal combinatorial patterned-batch codes (see Section I-E) corresponding to the case of servers with heterogeneous storage capacities as well. At the high level, the algorithm colors two sibling nodes at a time and proceeds recursively down to the two subtrees and repeats the process while maintaining the Ancestral Property: if a color is used for a node then it will not be used for the descendants of that node. Finding a way to split the colors for the two subtrees to guarantee that the algorithm runs smoothly until the end is a highly nontrivial task. Using our algorithm implemented in Java [7], we can generate a balanced ancestral coloring for the tree $T(30)$ (around two billion nodes) in under five minutes (with 16GB allocated for Java’s heap memory).

C. Application in Private Retrieval of Merkle Proofs

Privacy, which is a major feature required apart from scalability and security in the development of every blockchain network, may come in many different shapes and sizes (see, e.g., [8], [9], [10]). In this article, we are interested in the problem of a lightweight client (with a limited computation and storage capacity) retrieving a Merkle proof privately from a single full node to perform Simplified Payment Verification (SPV) [1]. Note that while we are using blockchains as a canonical application of ancestral colorings, our result applies to every context in which a Merkle tree is used and a Merkle proof needs to be retrieved privately, e.g., see [2], [3], [4], [5], [6]. We note that in [11], Lueks and
Goldberg also examined the same problem in the context of log servers storing web certificates. However, their solution relies on information-theoretic Private Information Retrieval schemes and assumes multiple non-colluding servers. In contrast, our solution is based on computational Private Information Retrieval schemes and requires a single server, thus avoiding the risk of server collusion. We postpone the detailed discussion of this problem till Section III and only give a high level explanation here on how ancestral colorings of trees can provide privacy in retrieving Merkle proofs.

Fig. 2. The nodes of a Merkle tree are stored at a blockchain full node and are partitioned into a number of parts/databases (ignoring the root) labeled with different colors. With a coloring that does not satisfy the Ancestral Property, privately retrieving individual nodes along the shaded root-to-leaf path (excluding the root, which is usually publicly known) may reveal the path: although the client can retrieve each node without letting the server know which one is being downloaded using a Private Information Retrieval scheme, it must query the green database twice, which allows the full node to identify this path because it is the only root-to-leaf path that contains two green nodes in the tree.

We observe that the hashes in a Merkle proof of a Merkle tree \[12\], treating as a perfect binary tree, correspond to tree nodes along a root-to-leaf path (excluding the root) in the swapped Merkle tree in which sibling nodes are swapped (see Section III-C for more details). Hence, the problem of privately retrieving a Merkle proof is equivalent to the problem of privately retrieving all nodes along a root-to-leaf path in a perfect binary tree. Our idea is to partition the nodes in the tree into \(h\) parts (a.k.a. color classes) and run \(h\) different Private Information Retrieval (PIR) schemes, e.g., SealPIR \[13\], \[14\], simultaneously on \(h\) CPU cores, treating each part as a separate database. Each PIR scheme allows the client to download one node/hash without revealing which node is being retrieved. An equivalent setting is to have \(h\) different servers, each storing a part of the tree and serving the client in parallel. The Balance Property of the partition/coloring guarantees that all databases are of roughly the same sizes (rounding up and down of \((2^h+1)/h\)) and hence all PIR schemes will take roughly the same time to finish, which is \(\Theta(2^{h+1}/h)\). Compared to the natural partition into \(h\) layers (the bottom one is of size \(2^h\)), our scheme is \(\Theta(h/2)\) times faster. The Ancestral Property of the coloring guarantees that all root-to-leaf paths are indistinguishable from each other. We demonstrate in Fig. 2 that a coloring violating the Ancestral Property may reveal a path if it has a different color pattern from the others, hence jeopardizing the privacy. If the coloring satisfies the Ancestral Property then each database/color class is queried exactly once, and it is impossible for the server to distinguish the request of one path from another. Our scheme, in a nutshell, is a combination of a PIR scheme and a novel optimal combinatorial patterned-batch code (see Section I-E), extending the idea proposed in \[15\] for regular batch codes.

D. A Connection to Equitable Coloring

We demonstrate below that by finding balanced ancestral colorings of perfect binary trees, we are also able to determine the equitable chromatic number of a new family of graphs.
An undirected graph $G$ (connected or not) is said to be equitable $k$-colorable if its vertices can be colored with $k$ colors such that no adjacent vertices have the same color and moreover, the sizes of the color classes differ by at most one [16]. The equitable chromatic number $\chi_e(G)$ is the smallest integer $k$ such that $G$ is equitable $k$-colorable. There have been extensive research in the literature on equitable coloring of graphs (see, for instance, [17], for a survey), noticeably, on the Equitable Coloring Conjecture, which states that for a connected graph $G$ that is neither a complete graph nor an odd cycle, then $\chi_e(G) \leq \Delta(G)$, where $\Delta(G)$ is the maximum degree of a vertex in $G$. Special families of graphs that support the Equitable Coloring Conjecture include trees, bipartite graphs, planar and outerplanar graphs, among others.

The existence of balanced ancestral colorings of the perfect binary tree $T(h)$ established in this work immediately implies that $\chi_e(T'(h)) \leq h$, where $T'(h)$ is a disconnected graph obtained from $T(h)$ by first removing its root and then by adding edges between every node $u$ and all of its descendants (see Fig. 3 for a toy example of $T'(3)$). Note that $T'(h)$ is neither a tree, a bipartite graph, a planar graph, nor an outerplanar graph for $h \geq 5$ (because it contains the complete graph $K_h$ as a subgraph). On top of that, the maximum vertex degree is $\Delta(T'(h)) = 2^h - 2$, which is very far from $h$ and therefore would a very loose bound on the equitable chromatic number of $T'(h)$.

Furthermore, as $h$ nodes along a path from node 2 (or 3) to a leaf are ancestors or descendants of each other (hence forming a complete subgraph $K_h$), they should all belong to different color classes, which implies that $\chi_e(T'(h)) \geq h$. Thus, $\chi_e(T'(h)) = h$. Moreover, a balanced ancestral coloring of $T(h)$ provides an equitable $h$-coloring of $T'(h)$. To the best of our knowledge, the equitable chromatic number of this family of graphs has never been discovered before. We will continue the discussion in Section II-C.

### E. A Connection to Combinatorial Batch Codes

The study of balanced ancestral colorings of perfect binary trees brings in two new dimensions to batch codes [15]: patterned batch retrieval (instead of arbitrary batch retrieval as often considered in the literature) and balanced storage capacity across servers (and generalized to heterogeneous storage capacity).

Given a set of $n$ distinct items, an $(n, N, h, m)$ combinatorial batch code (CBC) provides a way to assign $N$ copies of items to $m$ different servers, so that in order to retrieve a set of $h$ distinct items, an user can download at most one item from each server [15]. The goal is to minimize the total storage capacity required $N$. For instance, labelling $n = 7$ items from 1 to 7, and use $m = 5$ servers, an $(n = 7, N = 15, h = 5, m = 5)$ CBC allocates to five servers the sets $\{1, 6, 7\}, \{2, 6, 7\}, \{3, 6, 7\}, \{4, 6, 7\}, \{5, 6, 7\}$ (Paterson-Stinson-Wei [20]). One can verify that an arbitrary set of $h = 5$ items can be retrieved by collecting one item from each set. Here, the storage requirement is $N = 15 > n = 7$. It has been proved (see, e.g., [20]) that when $h = m$, the minimum possible $N$ is in the order of $\Theta(nh)$. That implies that the average replication factor across $n$ items is $\Theta(h)$.
Fig. 4. An illustration of a balanced combinatorial patterned-batch code with a minimum total storage overhead (no redundancy), which is spread out evenly among three servers. The set of items at each server corresponds to tree nodes in a color class of a balanced ancestral coloring of $T(3)$ as given in Fig. 1. A client only needs to download one item from each server for any batch request following a root-to-leaf-path pattern in $P = \{\{2, 4, 8\}, \{2, 4, 9\}, \{2, 5, 10\}, \{2, 5, 11\}, \{3, 6, 12\}, \{3, 6, 13\}, \{3, 7, 14\}, \{3, 7, 15\}\}$. 

It turns out that $N = n$ (equivalently, replication factor one or no replication) is achievable if the $h$ retrieved items are not arbitrary and follow a special pattern. More specifically, if the $n$ items can be organised as nodes in a perfect binary tree $T(h)$ except the root and only sets of nodes along root-to-leaf paths (ignoring the root) are to be retrieved, then each item only needs to be stored in one server and requires no replication across different servers, which implies that $N = n$. A trivial solution is to use layer-based sets, i.e., Server $i$ stores nodes in Layer $i$ of the tree, $i = 1, 2, \ldots, h$. If we also want to balance the storage across the servers, that is, to make sure that the number of items assigned to every server differs from each other by at most one, then the layer-based solution no longer works and a balanced ancestral coloring of the tree nodes (except the root) will be required (see Fig. 4 for an example). We refer to this kind of codes as balanced combinatorial patterned-batch codes. The balance requirement can also be generalized to the heterogeneous setting in which different servers may have different storage capacities. It is an intriguing open problem to discover other patterns (apart from the root-to-leaf paths considered in this paper) so that (balanced or heterogeneous) combinatorial patterned-batch codes with optimal total storage capacity $N = n$ exist.

The paper is organized as follows. We first introduce the Color-Splitting Algorithm and demonstrate its correctness in Section II. We then describe the application of balanced ancestral coloring of perfect binary trees in retrieving Merkle proofs privately and efficiently in Section III. We conclude the paper and discuss various open problems for future research in Section IV.

II. A Divide-And-Conquer Algorithm for Ancestral Colorings of Perfect Binary Trees

A. The Color-Splitting Algorithm

We first need the definition of a color sequence.

**Definition 1** (Color sequence). A color sequence of dimension $h$ is a sorted sequence of positive integers $\vec{c} = [c_1, \ldots, c_h]$, where $c_1 \leq c_2 \leq \cdots \leq c_h$. The sum $\sum_{i=1}^{h} c_i$ is referred to as the sequence’s total size. The element $c_i$ is called the color size, which represents the number of nodes in a tree that will be assigned Color $i$. The color sequence $\vec{c}$ is called balanced if the color sizes $c_i$ differ from each other by at most one, or equivalently, $c_j - c_i \leq 1$ for all $h \geq j > i \geq 1$. It is assumed that the total size of a color sequence is equal to the total number of nodes in a tree (excluding the root).

The Color-Splitting Algorithm (CSA) starts from a color sequence $\vec{c} = [c_1, \ldots, c_h]$ and proceeds to color the tree nodes, two sibling nodes at a time, from the top of the tree down to the bottom in a recursive manner while keeping track of the number of remaining nodes that can be assigned
Color \( i \). Note that the elements of a color sequence \( \vec{c} \) are always sorted in a non-decreasing order, e.g., \( \vec{c} = [4 \text{ Red}, 5 \text{ Green}, 5 \text{ Blue}] \), and CSA always try to color the children of the current root \( R \) with \textit{either} the same color \( 1 \) if \( c_1 = 2 \), \textit{or} with two different colors 1 and 2 if \( 2 < c_1 \leq c_2 \). The remaining color counts are split “evenly” between the left and the right subtrees of \( R \) while ensuring that the color used for each root’s child will no longer be used for the subtree rooted at that node to guarantee the Ancestral Property. The key technical point is to make sure that the split is done in a way that prevents the algorithm from getting stuck, i.e., to make sure that it always has enough colors to produce ancestral colorings for both subtrees. If a balanced ancestral coloring is required, CSA starts with the balanced color sequence \( \vec{c}^* = [c_1^*, \ldots, c_h^*] \), in which the color sizes \( c_i^* \)'s differ from each other by at most one.

Before introducing rigorous notations and providing a detailed algorithm description, let start with an example of how the algorithm works on \( T(3) \).

**Example 1.** The Color-Splitting Algorithm starts from the root node 1 with the balanced color sequence \( \vec{c}^* = [4, 5, 5] \), which means that it is going to assign Red (Color 1) to four nodes, Green (Color 2) to five nodes, and Blue (Color 3) to five nodes (see Fig. 5). We use \([4R, 5G, 5B]\) instead of \([4, 5, 5]\) to keep track of the colors. Note that the root needs no color. According to the algorithm’s rule, as Red and Green have lowest sizes, which are greater than two, the algorithm colors the left child (Node 2) red and the right child (Node 3) green. The dimension-3 color sequence \( \vec{c} = [4R, 5G, 5B] \) is then split into two dimension-2 color sequences \( \vec{a} = [2B, 4G] \) and \( \vec{b} = [3R, 3B] \). How the split works will be discussed in detail later, however, we can observe that both resulting color sequences have a valid total size 6 = 2 + 2\(^2\), which matches the number of nodes in each subtree. Moreover, as \( \vec{a} \) has no Red and \( \vec{b} \) has no Green, the Ancestral Property is guaranteed for Node 2 and Node 3, i.e., these two nodes have different colors from their descendants. The algorithm now repeats what it does to these two subtrees rooted at Node 2 and Node 3 using \( \vec{a} \) and \( \vec{b} \). For the left subtree rooted at 2, the color sequence \( \vec{a} = [2B, 4G] \) has two Blues, and so, according to CSA’s rule, the two children 4 and 5 of 2 both receive Blue as their colors. The remaining four Greens are split evenly into \([2G]\) and \([2G]\), which will then be used to color 8, 9, 10, and 11. The remaining steps are carried out in the same manner.

We observe that not every color sequence \( \vec{c} = [c_1, \ldots, c_h] \) of dimension \( h \), even when having a valid total size \( \sum_{i=1}^h c_i \), can be used to construct an ancestral coloring of the perfect binary tree \( T(h) \). For example, we can easily verify that there are no ancestral colorings of \( T(2) \) using \( \vec{c} = [1, 5] \), i.e., 1 Red and 5 Greens, and similarly, no ancestral colorings of \( T(3) \) using \( \vec{c} = [2, 3, 9] \), i.e., 2 Reds, 3
Definition 2 (Feasible color sequence). A (sorted) color sequence $\vec{c}$ of dimension $h$ is called $h$-feasible if it satisfies the following two conditions:

- (C1) $\sum_{i=1}^{\ell} c_i \geq \sum_{i=1}^{h} 2^i$, for every $1 \leq \ell \leq h$, and
- (C2) $\sum_{i=1}^{h} c_i = \sum_{i=1}^{h} 2^i = 2^{h+1} - 2$.

Condition (C1) means that Colors 1, 2, ..., $\ell$ are sufficient in numbers to color all nodes in Layers 1, 2, ..., $\ell$ of the perfect binary tree $T(h)$ (Layer $i$ has $2^i$ nodes). Condition (C2) states that the total size of $c$ is equal to the number of nodes in $T(h)$.

The reader may notice that the conditions (C1) and (C2) correspond to the well-known concept of majorization. We will discuss this interesting connection in Section II-C.

Example 2. The following color sequences (taken from the actual coloring of nodes in the trees $T(1), T(2), T(3)$ in Fig. 1) are feasible: [2], [3, 3], and [4, 5, 5]. The sequences [2, 3] and [3, 4, 6, 17] are not feasible: $2 + 3 < 6 = 2 + 2^2$, $3 + 4 + 6 < 14 = 2 + 2^2 + 2^3$. Clearly, color sequences of the forms $[1, \ldots, 2, 2, \ldots, 2]$, or $[2, 3, \ldots, 3]$, or $[3, 4, 6, \ldots]$ are not feasible due to the violation of (C1).

Definition 3. The perfect binary $T(h)$ is said to be ancestral $\vec{c}$-colorable, where $\vec{c} = [c_1, \ldots, c_h]$ is a color sequence, if there exists an ancestral coloring of $T(h)$ in which precisely $c_i$ nodes are assigned Color $i$, for all $i = 1, \ldots, h$. Such a coloring is called an ancestral $\vec{c}$-coloring of $T(h)$.

The following lemma states that every ancestral coloring for $T(h)$ requires at least $h$ colors.

Lemma 1. If the perfect binary tree $T(h)$ is ancestral $\vec{c}$-colorable, where $\vec{c} = [c_1, \ldots, c_h]$, then $h' \geq h$. Moreover, if $h' = h$ then all $h$ colors must show up on nodes along any root-to-leaf path (except the root, which is colorless). Equivalently, nodes having the same color $i \in \{1, 2, \ldots, h\}$ must collectively belong to $2^h$ different root-to-leaf paths.

Proof. A root-to-leaf path contains exactly $h$ nodes except the root. Since these nodes are all ancestors and descendants of each other, they should have different colors. Thus, $h' \geq h$. As $T(h)$ has precisely $2^h$ root-to-leaf paths, other conclusions follow trivially.

Theorem 1 characterizes all color sequences of dimension $h$ that can be used to construct an ancestral coloring of $T(h)$. Note that the balanced color sequence that corresponds to a balanced ancestral coloring is only a special case among all such sequences.

Theorem 1 (Ancestral-Coloring Theorem for Perfect Binary Trees). For every $h \geq 1$ and every color sequence $\vec{c} = [c_1, \ldots, c_h]$ of dimension $h$, the perfect binary tree $T(h)$ is ancestral $\vec{c}$-colorable if and only if $\vec{c}$ is $h$-feasible.

Proof. The Color-Splitting Algorithm can be used to show that if $\vec{c}$ is $h$-feasible then $T(h)$ is ancestral $\vec{c}$-colorable. For the necessary condition, we show that if $T(h)$ is ancestral $\vec{c}$-colorable then $\vec{c}$ must be $h$-feasible. Indeed, for each $1 \leq \ell \leq h$, in an ancestral $\vec{c}$-coloring of $T(h)$, the nodes having the same color $i$, $1 \leq i \leq \ell$, should collectively belong to $2^h$ different root-to-leaf paths in the tree according to Lemma 1. Note that each node in Layer $i$, $i = 1, 2, \ldots, h$, belongs to $2^{h-i}$ different paths. Therefore, collectively, nodes in Layers 1, 2, ..., $\ell$ belong to $\sum_{i=1}^{\ell} 2^i \times 2^{h-i} = \ell 2^h$ root-to-leaf paths. We can see that each path is counted $\ell$ times in this calculation. Note that each node in Layer $i$ belong to strictly more paths than each node in Layer $j$ if $i < j$. Therefore, if (C1) is violated, i.e., $\sum_{i=1}^{\ell} c_i < \sum_{i=1}^{\ell} 2^i$, which implies that the number of nodes having colors 1, 2, ..., $\ell$ is smaller than the total number of nodes in Layers 1, 2, ..., $\ell$, then the total number of paths (each can be counted more than once) that nodes having colors 1, 2, ..., $\ell$ belong to is strictly smaller than $\ell 2^h$. As a consequence, there exists a color $i \in \{1, 2, \ldots, \ell\}$ such that nodes having this color collectively belong to fewer than $2^h$ paths, contradicting Lemma 1.

\[ \square \]
**Corollary 1.** A balanced ancestral coloring exists for the perfect binary tree $T(h)$ for every $h \geq 1$.

**Proof.** This follows directly from Theorem 1 noting that a balanced color sequence of dimension $h$ is also $h$-feasible (see Corollary 2 proved in the next section).

We now describe formally the Color-Splitting Algorithm. The algorithm starts at the root $R$ of $T(h)$ with an $h$-feasible color sequence $\vec{c} = [c_1, \ldots, c_h]$ (see ColorSplitting($h, \vec{c}$)) and then colors the two children $A$ and $B$ of the root as follows: these nodes receive the same Color 1 if $c_1 = 2$ or Color 1 and Color 2 if $c_1 > 2$. Next, the algorithm splits the remaining colors in $\vec{c} = [c_1, \ldots, c_h]$ (whose total size has already been reduced by two) into two $(h - 1)$-feasible color sequences $\vec{a}$ and $\vec{b}$, which are subsequently used for the two subtrees $T(h - 1)$ rooted at $A$ and $B$ (see ColorSplittingRecursive($R, h, \vec{c}$)). Note that the splitting rule (see FeasibleSplit($h, \vec{c}$)) ensures that if Color $i$ is used for a node then it will no longer be used in the subtree rooted at that node, hence guaranteeing the Ancestral Property. We prove in Section II-B that it is always possible to split an $h$-feasible sequence into two new $(h - 1)$-feasible sequences, which guarantees a successful termination if the input of the CSA is an $h$-feasible color sequence.

**Algorithm ColorSplitting($h, \vec{c}$)**

```
// Given a feasible color sequence $\vec{c} = [c_1, \ldots, c_h]$,  
// the algorithm finds an ancestral $\vec{c}$-coloring of $T(h)$ 
Set $R := 1$; // the root of $T(h)$ is 1, which requires no color 
ColorSplittingRecursive($R, h, \vec{c}$);
```

**Procedure ColorSplittingRecursive($R, h, \vec{c}$)**

```
// $R$ is the root node of the current subtree $T(h)$ of height $h$  
// Either $R$ needs no color ($R = 1$) or $R$ has already been colored in the previous call  
// $\vec{c} = [c_1, \ldots, c_h]$ is a feasible color sequence, which implies that $2 \leq c_1 \leq c_2 \leq \cdots \leq c_h$  
// This procedure colors the two children of $R$ and create feasible color sequences  
// for its left and right subtrees  
if $h \geq 1$ then
    $A := 2R; B := 2R + 1$; // left and right child of $R$
    if $c_1 = 2$ then
        Assign Color 1 to both $A$ and $B$;
    else
        Assign Color 1 to $A$ and Color 2 to $B$;
    end if
else
    if $h \geq 2$ then
        Let $\vec{a}$ and $\vec{b}$ be the output color sequences of FeasibleSplit($h, \vec{c}$);
        ColorSplittingRecursive($A, h - 1, \vec{a}$);
        ColorSplittingRecursive($B, h - 1, \vec{b}$);
    end if
end if
```

**Example 3.** We illustrate the Color-Splitting Algorithm when $h = 4$ in Fig. 6. The algorithm starts with a 4-feasible sequence $\vec{c} = [3, 6, 8, 13]$, which corresponds to 3 Reds, 6 Greens, 8 Blues, and 13 Purples. The root node 1 doesn’t required any color. As $2 < c_1 = 3 < c_2 = 8$, CSA colors 2 with Red, 3 with Green, and splits $\vec{c}$ into $\vec{a} = [3B, 5G, 6P]$ and $\vec{b} = [2R, 5B, 7P]$, which are both 3-feasible and will be used to color the subtrees rooted at 2 and 3, respectively. Note that $\vec{a}$ has no Red and $\vec{b}$ has no Green, which enforces the Ancestral Property for Node 2 and Node 3.
Indeed, according to the coloring rule in the algorithm colors the left and right children of the root node \( R \). As the algorithm proceeds in a recursive manner, it suffices to show that at each step, after the coloring, the Color-Splitting Algorithm, if terminates successfully, will generate an ancestral coloring.

Procedure FeasibleSplit\((h, \vec{c})\)

```plaintext
// This algorithm splits a (sorted) \( h \)-feasible sequence into two (sorted) \((h-1)\)-feasible ones, // which will be used for coloring the subtrees, // only works when \( h \geq 2 \)
// Case 1: \( c_1 = 2 \), note that \( c_1 \geq 2 \) due to the feasibility of \( \vec{c} \)
if \( c_1 = 2 \) then
    Set \( a_2 := \lfloor c_2/2 \rfloor \) and \( b_2 := \lceil c_2/2 \rceil \);
    Set \( S_2(a) := a_2 \) and \( S_2(b) := b_2 \); // note that \(|S_i(a) - S_i(b)| \leq 1 \) for \( 2 \leq i \leq h \)
    for \( i = 3 \) to \( h \) do
        if \( S_{i-1}(a) < S_{i-1}(b) \) then
            Set \( a_i := \lceil c_i/2 \rceil \) and \( b_i := \lfloor c_i/2 \rfloor \);
        else
            Set \( a_i := \lfloor c_i/2 \rfloor \) and \( b_i := \lceil c_i/2 \rceil \);
        end if
        Update \( S_i(a) := S_{i-1}(a) + a_i \) and \( S_i(b) := S_{i-1}(b) + b_i \);
    end for
// Case 2: \( c_1 > 2 \)
else
    Set \( a_2 := c_2 - 1 \) and \( b_2 := c_1 - 1 \); // \( b_2 \) now refers to the number of colors 1 in \( \vec{b} \)
    if \( h \geq 3 \) then
        Set \( a_3 := \lfloor \frac{c_2+c_3-c_2}{2} \rfloor \) and \( b_3 := c_2 - c_1 + \lfloor \frac{c_2+c_3-c_2}{2} \rfloor \);
        Set \( S_3(a) := a_2 + a_3 \) and \( S_3(b) := b_2 + b_3 \); // \( \vec{b} \) note that \(|S_i(a) - S_i(b)| \leq 1 \) for \( 3 \leq i \leq h \)
        for \( i = 4 \) to \( h \) do
            if \( S_{i-1}(a) < S_{i-1}(b) \) then
                Set \( a_i := \lceil c_i/2 \rceil \) and \( b_i := \lfloor c_i/2 \rfloor \);
            else
                Set \( a_i := \lfloor c_i/2 \rfloor \) and \( b_i := \lceil c_i/2 \rceil \);
            end if
            Update \( S_i(a) := S_{i-1}(a) + a_i \) and \( S_i(b) := S_{i-1}(b) + b_i \);
        end for
    end if
end if
Sort \( \vec{a} = [a_2, a_3, \ldots, a_h] \) and \( \vec{b} = [b_2, b_3, \ldots, b_h] \) in non-decreasing order;
return \( \vec{a} \) and \( \vec{b} \);
```

B. A Proof of Correctness

Lemma 2. The Color-Splitting Algorithm, if terminates successfully, will generate an ancestral coloring.

Proof. As the algorithm proceeds in a recursive manner, it suffices to show that at each step, after the algorithm colors the left and right children \( A \) and \( B \) of the root node \( R \), it will never use the color assigned to \( A \) for any descendant of \( A \) nor use the color assigned to \( B \) for any descendant of \( B \). Indeed, according to the coloring rule in ColorSplittingRecursive\((R, h, \vec{c})\) and FeasibleSplit\((h, \vec{c})\), if \( \vec{c} = [c_1, \ldots, c_h] \) is the color sequence available at the root \( R \) and \( c_1 = 2 \), then after allocating Color 1 to both \( A \) and \( B \), there will be no more Color 1 to assign to any node in both subtrees rooted at \( A \) and \( B \), and hence, our conclusion holds. Otherwise, if \( 2 < c_1 \leq c_2 \), then \( A \) is assigned Color 1 and according to FeasibleSplit\((h, \vec{c})\), the color sequence \( \vec{a} = [a_2 = c_2 - 1, a_3, \ldots, a_h] \) used to color the descendants of \( A \) no longer has Color 1. The same argument works for \( B \) and its descendants. In other words, in this case, Color 1 is used exclusively for \( A \) and descendants of \( B \) while Color 2 is used exclusively for \( B \) and descendants of \( A \), which will ensure the Ancestral Property for both \( A \) and \( B \).
We establish the correctness of the Color-Splitting Algorithm in Lemma 3.

**Lemma 3 (Correctness of Color-Splitting Algorithm).** *If the initial input color sequence \( \vec{c} \) is \( h \)-feasible then the Color-Splitting Algorithm terminates successfully and generates an ancestral \( \vec{c} \)-coloring for \( T(h) \). Its time complexity is \( O(2^{h+1} \log h) \), almost linear in the number of tree nodes.*

**Proof.** Note that because the Color-Splitting Algorithm starts off with the desirable color sequence \( \vec{c} \), if it terminates successfully then the output coloring, according to Lemma 2, will be the desirable ancestral \( \vec{c} \)-coloring. Therefore, our remaining task is to show that the Color-Splitting Algorithm always terminates successfully. Both \( \text{ColorSplittingRecursive}(R, h, \vec{c}) \) and \( \text{FeasibleSplit}(h, \vec{c}) \) work if \( c_1 \geq 2 \) when \( h \geq 1 \), i.e., when the current root node still has children below. This condition holds trivially in the beginning because the original color sequence is feasible. To guarantee that \( c_1 \geq 2 \) in all subsequent algorithm calls, we need to prove that starting from an \( h \)-feasible color sequence \( \vec{c} \) with \( h \geq 2 \), \( \text{FeasibleSplit}(h, \vec{c}) \) always produces two \( (h-1) \)-feasible color sequences \( \vec{a} \) and \( \vec{b} \) for the two subtrees. This is the most technical and lengthy part in the proof of correctness of CSA and will be settled separately in Lemma 5 and Lemma 6.

In the remainder of the proof of this lemma, we analyze the time complexity of CSA. Let \( C(n) \), \( n = 2^h \), be the number of basic operations (e.g., assignments) required by the recursive procedure \( \text{ColorSplittingRecursive}(R, h, \vec{c}) \). Then from its description, the following recurrence relation holds

\[
C(n) = \begin{cases} 
2C(n/2) + \alpha h \log h, & \text{if } n \geq 4, \\
\beta, & \text{if } n = 2, 
\end{cases}
\]

where \( \alpha \) and \( \beta \) are positive integer constants and \( \alpha h \log h \) is the running time of \( \text{FeasibleSplit}(h, \vec{c}) \) (dominated by the time required for sorting \( \vec{a} \) and \( \vec{b} \)), noting that \( h = \log_2 n \). We can apply the backward substitution method (or an induction proof) to determine \( C(n) \) as follows.
\[ C(n) = 2C(n/2) + \alpha h \log h \]
\[
\begin{align*}
&= 2(2C(n/2^2) + \alpha(h - 1) \log(h - 1)) + \alpha h \log h \\
&= 2^2C(n/2^2) + \alpha(2(h - 1) \log(h - 1) + h \log h) \\
&= 2^2(2C(n/2^3) + \alpha(h/2) \log(h/2)) + \alpha(2(h - 1) \log(h - 1) + h \log h) \\
&= 2^3C(n/2^3) + \alpha(2^2(h - 2) \log(h - 2) + 2(h - 1) \log(h - 1) + h \log h) \\
&= \cdots \\
&= 2^kC(n/2^k) + \alpha \sum_{i=0}^{k-1} 2^i(h - i) \log(h - i),
\end{align*}
\]

for every \(1 \leq k \leq h - 1\). Substituting \(k = h - 1\) in the above equality and noting that \(C(2) = \beta\), we obtain
\[
\begin{align*}
C(n) &= \beta 2^{h-1} + \alpha \sum_{i=0}^{h-1} 2^i(h - i) \log(h - i) \\
&\leq \beta 2^{h-1} + \alpha \left( h \sum_{i=0}^{h-1} 2^i - \sum_{i=0}^{h-1} i2^i \right) \log h \\
&= \beta 2^{h-1} + \alpha (h(2^h - 1) - ((2h)/2^h + 2)) \log h = \beta 2^{h-1} + \alpha(2^{h+1} - h + 2) \log h \\
&\in O(2^{h+1} \log h) = O(n \log \log n).
\end{align*}
\]

Therefore, the Color-Splitting Algorithm has a running time \(O(2^{h+1} \log h)\) as claimed. \(\square\)

To complete the proof of correctness for CSA, it remains to settle Lemma 5 and Lemma 6 which state that the procedure FeasibleSplit \((h, \vec{c})\) produces two \((h - 1)\)-feasible color sequences \(\vec{a}\) and \(\vec{b}\) from a \(h\)-feasible color sequence \(\vec{c}\). To this end, we first establish Lemma 4 which provides a crucial step in the proofs of Lemma 5 and Lemma 6. In essence, it establishes that if Condition (C1) in the feasibility definition (see Definition 2) is satisfied at the two indices \(m\) and \(\ell\) of a non-decreasing sequence \(a_1, \ldots, a_m, \ldots, a_\ell\) where \(m < \ell\), and moreover, the elements \(a_{m+1}, a_{m+2}, \ldots, a_\ell\) differ from each other by at most one, then (C1) also holds at every middle index \(p\) \((m < p < \ell)\). This simplifies significantly the proof that the two color sequences \(\vec{a}\) and \(\vec{b}\) generated from an \(h\)-feasible color sequence \(\vec{c}\) in the Color-Splitting Algorithm also satisfy (C1) (replacing \(h\) by \(h - 1\)), and hence, are \((h - 1)\)-feasible.

**Lemma 4.** Suppose \(0 \leq m < \ell\) and the sorted sequence of integers \((a_1, \ldots, a_\ell)\),
\[
2 \leq a_1 \leq \cdots \leq a_m \leq a_{m+1} \leq \cdots \leq a_\ell,
\]
satisfies the following properties, in which \(S_x = \sum_{i=1}^x a_i\),

- (P1) \(S_m \geq \sum_{i=1}^m 2^i\) (trivially holds if \(m = 0\) since both sides of the inequality will be zero),
- (P2) \(S_\ell \geq \sum_{i=1}^\ell 2^i\),
- (P3) \(a_{m+1} = \cdots = a_{m+k} = \lceil \frac{c}{2} \rceil\), \(a_{m+k+1} = \cdots = a_\ell = \lceil \frac{c}{2} \rceil\), for some \(0 \leq k \leq \ell - m\) and \(c \geq 4\).

Then \(S_p \geq \sum_{i=1}^p 2^i\) for every \(p\) satisfying \(m < p < \ell\).

**Proof.** We consider two cases based on the parity of \(c\).

**Case 1: \(c\) is even.** In this case, from (P3) we have \(a_{m+1} = \cdots = a_\ell = a\), where \(a \leq c/2\).

As \(p > m\), we have \(S_p = S_m + \sum_{i=m+1}^p a_i\). If \(S_m \geq \sum_{i=1}^p 2^i\) then the conclusion trivially holds. Otherwise, let \(S_m = \sum_{i=1}^p 2^i - \delta\), for some \(\delta > 0\). Then (P1) implies that
\[
0 < \delta = \sum_{i=1}^p 2^i - S_m \leq \sum_{i=1}^p 2^i - \sum_{i=1}^m 2^i = \sum_{i=m+1}^p 2^i,
\]
(1)

Moreover, from (P2) we have
\[
S_\ell = S_m + (\ell - m)a = \left( \sum_{i=1}^p 2^i - \delta \right) + (\ell - m)a \geq \sum_{i=1}^\ell 2^i,
\]

which implies that
\[ a \geq \frac{1}{\ell - m} \left( \sum_{i=p+1}^{\ell} 2^i + \delta \right). \]

Therefore,
\[ S_p = S_m + (p - m)a = \left( \sum_{i=1}^{p} 2^i - \delta \right) + (p - m)a \geq \left( \sum_{i=1}^{p} 2^i - \delta \right) + \frac{p - m}{\ell - m} \left( \sum_{i=p+1}^{\ell} 2^i + \delta \right). \]

Hence, in order to show that \( S_p \geq \sum_{i=1}^{p} 2^i \), it suffices to demonstrate that
\[ (p - m) \sum_{i=p+1}^{\ell} 2^i \geq (\ell - p)\delta. \]

Making use of (1), we need to show that the following inequality holds
\[ (p - m) \sum_{i=p+1}^{\ell} 2^i \geq (\ell - p) \sum_{i=m+1}^{p} 2^i, \]
which is equivalent to
\[ \frac{\sum_{i=p+1}^{\ell} 2^i}{\sum_{i=m+1}^{p} 2^i} \geq \frac{\ell - p}{p - m} \iff \frac{2^{p-m} (2^{\ell-p} - 1)}{2^{p-m} - 1} \geq \frac{\ell - p}{p - m}. \]

The last inequality holds because \( \frac{2^{p-m}}{2^{p-m} - 1} > 1 \) and \( 2^{\ell-p} - 1 \geq \ell - p \) for all \( 0 \leq m < p < \ell \). Here we use the fact that \( 2^x - 1 - x \geq 0 \) for all \( x \geq 1 \). This establishes Case 1.

**Case 2:** \( \ell \) is odd. In this case, from (P3) we have \( a_{m+1} = \ldots = a_{m+k} = a \) and \( a_{m+k+1} = \ldots = a_\ell = a + 1 \), where \( a \triangleq \lfloor c/2 \rfloor \). We claim that if the inequality
\[ S_{m+k} \geq \sum_{i=1}^{m+k} 2^i \]  
holds then we can reduce Case 2 to Case 1. Indeed, suppose that (2) holds. Then by replacing \( \ell \) by \( m + k \) and applying Case 1, we deduce that \( S_p \geq \sum_{i=1}^{p} 2^i \) for every \( p \) satisfying \( m < p < m + k \). Similarly, by replacing \( m \) by \( m + k \) and applying Case 1, the inequality \( S_p \geq \sum_{i=1}^{p} 2^i \) holds for every \( p \) satisfying \( m + k < p < \ell \). Thus, in the remainder of the proof, we aim to show that (2) is correct.

To simplify the notation, set \( p \triangleq m + k \). If \( p = m \) or \( p = \ell \) then (P1) and (P2) imply (2) trivially. Thus, we assume that \( m < p < \ell \). Since \( S_p = S_m + \sum_{i=m+1}^{p} a_i \), if \( S_m \geq \sum_{i=1}^{p} 2^i \) then the conclusion trivial holds. Therefore, let \( S_m = \sum_{i=1}^{p} 2^i - \delta \), for some \( \delta > 0 \). Similar to Case 1, (P1) implies that
\[ 0 < \delta = \sum_{i=1}^{p} 2^i - S_m \leq \sum_{i=1}^{p} 2^i - \sum_{i=1}^{m} 2^i = \sum_{i=m+1}^{p} 2^i. \]  

Moreover, from (P2) we have
\[ S_\ell = S_m + (\ell - m)a + (\ell - p) = \left( \sum_{i=1}^{p} 2^i - \delta \right) + (\ell - m)a + (\ell - p) \geq \sum_{i=1}^{\ell} 2^i, \]
which implies that
\[ a \geq \frac{1}{\ell - m} \left( \sum_{i=p+1}^{\ell} 2^i + \delta - (\ell - p) \right). \]

Therefore,
\[ S_p = S_m + (p - m)a = \left( \sum_{i=1}^{p} 2^i - \delta \right) + (p - m)a \geq \left( \sum_{i=1}^{p} 2^i - \delta \right) + \frac{p - m}{\ell - m} \left( \sum_{i=p+1}^{\ell} 2^i + \delta - (\ell - p) \right). \]
Hence, in order to show that \( S_p \geq \sum_{i=1}^{p} 2^i \), it suffices to demonstrate that
\[
(p - m) \left( \sum_{i=p+1}^{\ell} 2^i - (\ell - p) \right) \geq (\ell - p)\delta.
\]
Making use of (3), we need to show that the following inequality holds
\[
(p - m) \left( \sum_{i=p+1}^{\ell} 2^i - (\ell - p) \right) \geq (\ell - p) \sum_{i=m+1}^{p} 2^i,
\]
which is equivalent to
\[
\frac{\sum_{i=p+1}^{\ell} 2^i - (\ell - p)}{\sum_{i=m+1}^{p} 2^i} \geq \frac{\ell - p}{p - m} \iff \frac{2^{p-m} (2^{\ell-p} - 1) - (\ell - p)2^{m+1}}{2^{p-m} - 1} \geq \frac{\ell - p}{p - m}.
\]
If \( \ell - p = 1 \) then the last inequality of (4) becomes
\[
\frac{2^{p-m} - 1/2^{m+1}}{2^{p-m} - 1} \geq \frac{1}{p - m},
\]
which is correct because \( 1/2^{m+1} < 1 \) and \( p - m \geq 1 \). If \( \ell - p \geq 2 \) then the last inequality of (4) can be rewritten as
\[
\frac{2^{p-m} - 1}{2^{p-m} - 1} \geq \frac{2^{\ell-p} - 1 - \frac{\ell - p}{2^{\ell-p} - 1}}{\ell - p} \geq \frac{\ell - p}{p - m},
\]
which is correct because \( \frac{2^{p-m}}{2^{p-m} - 1} > 1 \geq \frac{1}{p - m} \) for \( p > m \) and
\[
2^{\ell-p} - 1 - \frac{\ell - p}{2^{\ell-p} - 1} \geq 2^{\ell-p} - 1 - \frac{\ell - p}{4} \geq \ell - p,
\]
noting that \( p \geq 1 \) and that \( 2^{x} - 1 - \frac{5}{4}x > 0 \) for every \( x \geq 2 \). This establishes Case 2. \( \square \)

As a corollary of Lemma 4, a balanced color sequence is feasible.

**Corollary 2 (Balanced color sequence).** For \( h \geq 1 \), set \( u = (2^h + 1 - 2) \) mod \( h \) and \( \overline{c} = [c_1, c_2, \ldots, c_h] \), where \( c_i = \left\lfloor \frac{2^{h+1} - 2}{h} \right\rfloor \) if \( 1 \leq i < h - u \) and \( c_i = \left\lfloor \frac{2^{h+1} - 2}{h} \right\rfloor \) for \( h - u + 1 \leq i \leq h \). Then \( \overline{c} \), referred to as the (sorted) balanced sequence, is \( h \)-feasible.

**Proof.** The proof of this lemma follows by applying Lemma 4 with \( m = 0 \). \( \square \)

Lemma 5 and Lemma 6 establish that as long as \( \overline{c} \) is an \( h \)-feasible color sequence, FeasibleSplit\((h, \overline{c})\) will produce two \((h - 1)\)-feasible color sequences that can be used in the next calls of ColorSplittingRecursive(). The two lemmas tackle the case \( c_1 = 2 \) and \( c_1 > 2 \), respectively.

**Lemma 5.** Suppose \( \overline{c} = [c_1, \ldots, c_h] \) is an \( h \)-feasible color sequence, where \( 2 = c_1 \leq c_2 \leq \cdots \leq c_h \). Let \( \overline{a} \) and \( \overline{b} \) be two sequences of dimension \( h - 1 \) obtained from \( \overline{c} \) as in FeasibleSplit\((h, \overline{c})\) Case 1, i.e., \( a_2 := [c_2/2] \) and \( b_2 := [c_2/2] \), and for \( i = 3, 4, \ldots, h \),
\[
\begin{aligned}
\{ a_i := [c_i/2] \text{ and } b_i := [c_i/2] \}, & \quad \text{ if } \sum_{j=2}^{i-1} a_j < \sum_{j=2}^{i-1} b_j, \\
\{ a_i := [c_i/2] \text{ and } b_i := [c_i/2] \}, & \quad \text{ otherwise.}
\end{aligned}
\]
Then \( \overline{a} \) and \( \overline{b} \), after being sorted, are \((h - 1)\)-feasible color sequences. We assume that \( h \geq 2 \).

**Proof.** To make the proof more readable, let \( \overline{a} = [a_2', \ldots, a_h'] \) and \( \overline{b} = [b_2', \ldots, b_h'] \) be obtained from \( \overline{a} \) and \( \overline{b} \) after their elements are sorted in non-decreasing order. According to Definition 2, the goal is to show that (C1) and (C2) hold for \( \overline{a}' \) and \( \overline{b}' \), while replacing \( h \) by \( h - 1 \). Because the sums \( S_i(a) \triangleq \sum_{j=2}^{i} a_j \) and \( S_i(b) \triangleq \sum_{j=2}^{i} b_j \) always differ from each other by at most one for every \( 2 \leq i \leq h \), it is clear that at the end when \( i = h \), they should be the same and equal to half of \( S_h(c) \triangleq \sum_{j=2}^{h} c_j = 2^h \sum_{j=2}^{h-1} 2^j = 2(\sum_{j=2}^{h-1} 2^j) \). Therefore, (C2) holds for \( \overline{a}' \) and \( \overline{b}' \) and hence, for \( \overline{a} \) and \( \overline{b} \) as well. It remains to show that (C1) holds for these two color sequences, that is, to prove
that $\sum_{i=2}^{\ell} a_i' \geq \sum_{i=1}^{\ell-1} 2^i$ and $\sum_{i=2}^{\ell} b_i' \geq \sum_{i=1}^{\ell-1} 2^i$ for every $2 \leq \ell \leq h$ (note that $\vec{a}'$ and $\vec{b}'$ both start from index 2).

Since $a_i$ and $b_i$ are assigned either $\lfloor c_i/2 \rfloor$ or $\lceil c_i/2 \rceil$ in somewhat an alternating manner, which keeps the sums $S_i(a) \triangleq \sum_{j=2}^{i} a_i$ and $S_i(b) \triangleq \sum_{j=2}^{i} b_i$ differ from each other by at most one, it is obvious that each sum will be approximately half of $\sum_{j=2}^{i} c_j$ (rounded up or down), which is greater than or equal to $\sum_{j=2}^{i} 2^j = 2 \sum_{j=1}^{i-1} 2^j$. Therefore, (C1) holds for $\vec{a}$ and $\vec{b}$. However, the trouble is that this may no longer be true after sorting, in which smaller values are shifted to the front. We show below that (C1) still holds for $\vec{a}'$ and $\vec{b}'$ using Lemma 4.

As $\vec{c}$ is a sorted sequence, we can partition $c_2, c_3, \ldots, c_h$ into $k$ different runs where within each run all $c_i$'s are equal,

$$c_2 = \cdots = c_i < c_{i+1} = \cdots = c_{i+q} < c_{i+q+1} = \cdots = c_k \equiv c_h.$$

For $r = 1, 2, \ldots, k$, let $R_r \triangleq [i_r-1 + 1, i_r]$, where $i_0 \triangleq 1$. Then (5) means that for each $r \in [k]$, $c_i$'s are the same for all $i \in R_r$ and moreover, $c_i < c_i'$ if $i \in R_r$, $i' \in R_{r'}$, and $r < r'$. In order to show that (C1) holds for $\vec{a}'$ and $\vec{b}'$, our strategy is to first prove that (C1) holds for $\vec{a}'$ and $\vec{b}'$ at the end-points $\ell = i_r$ of the runs $R_r$, $r \in [k]$, and then employ Lemma 4 to conclude that (C1) also holds for these color sequences at all the middle-points of the runs.

Since $\lfloor c_i/2 \rfloor \leq a_i \leq \lceil c_i/2 \rceil$ for every $2 \leq i \leq h$, it is clear that $a_i \leq a_i'$ if $i \in R_r$, $i' \in R_{r'}$, and $r < r'$. Therefore, $\vec{a}'$ can be obtained from $\vec{a}$ by sorting its elements locally within each run. As a consequence, $\sum_{i \in R_r} a_i' = \sum_{i \in R_r} a_i$ for every $r \in [k]$, which implies that

$$S_i(a') \triangleq \sum_{i=2}^{i_r} a_i' = \sum_{i=2}^{i_r} a_i \geq \sum_{i=1}^{i_r-1} 2^i,$$

where the last inequality comes from the fact that (C1) holds for $\vec{a}$ and $\vec{b}$ (as shown earlier). The inequality (6) implies that (C1) holds for $\vec{a}'$ at the end-points $i_r$, $r \in [k]$, of the runs. Applying Lemma 4, we deduce that (C1) also holds for $\vec{a}'$ at every middle index $p \in R_r$ for all $r \in [k]$. Therefore, (C1) holds for $\vec{a}'$, and similarly, for $\vec{b}'$. This completes the proof.

Lemma 6. Suppose $\vec{c} = [c_1, \ldots, c_h]$ is an $h$-feasible color sequence, where $2 < c_1 \leq c_2 \leq \cdots \leq c_h$. Let $\vec{a}$ and $\vec{b}$ be two sequences of dimension $h-1$ obtained from $\vec{c}$ as in FeasibleSplit$(h, \vec{c})$ Case 2, i.e., $a_2 := c_2 - 1$ and $b_2 := c_1 - 1$, and if $h \geq 3$ then

$$a_3 := \left\lfloor \frac{c_3 + c_1 - c_2}{2} \right\rfloor, \quad b_3 := c_2 - c_1 + \left\lfloor \frac{c_3 + c_1 - c_2}{2} \right\rfloor,$$

and for $i = 4, 5, \ldots, h$,

$$a_i := \lceil c_i/2 \rceil \text{ and } b_i := \lfloor c_i/2 \rfloor, \quad \text{if } \sum_{j=2}^{i-1} a_j < \sum_{j=2}^{i-1} b_j,$$

$$a_i := \lfloor c_i/2 \rfloor \text{ and } b_i := \lceil c_i/2 \rceil, \quad \text{otherwise}.$$

Then $\vec{a}$ and $\vec{b}$, after being sorted, are $(h-1)$-feasible color sequences. Note that while $a_i$ ($2 \leq i \leq h$) and $b_i$ ($3 \leq i \leq h$) correspond to Color $i$, $b_2$ actually corresponds to Color 1. We assume that $h \geq 2$.

Proof. See Appendix A.

C. A Connection to the Theory of Majorization

We discuss below the connection of some of our results to the theory of majorization [21].

Definition 4 (Majorization). For two vectors $\vec{x} = [x_1, \ldots, x_h]$ and $\vec{y} = [y_1, \ldots, y_h]$ in $\mathbb{R}^h$ with $x_1 \geq \ldots \geq x_h$ and $y_1 \geq \ldots \geq y_h$, $\vec{y}$ majorizes $\vec{x}$, denoted by $\vec{x} \prec \vec{y}$, if the following conditions hold.

- $\sum_{i=1}^{\ell} y_i \geq \sum_{i=1}^{\ell} x_i$, for every $1 \leq \ell \leq h$, and
- $\sum_{i=1}^{h} x_i = \sum_{i=1}^{h} y_i$.

Equivalently, when $\vec{x}$ and $\vec{y}$ are sorted in non-decreasing order, i.e., $x_1 \leq \ldots \leq x_h$ and $y_1 \leq \ldots \leq y_h$, $\vec{x} \prec \vec{y}$ if the following conditions hold.
• $\sum_{i=1}^{\ell} y_i \leq \sum_{i=1}^{\ell} x_i$, for every $1 \leq \ell \leq h$, and
• $\sum_{i=1}^{h} y_i = \sum_{i=1}^{h} x_i$.

Now, let $\vec{c}_h = [2, 2^2, \ldots, 2^h]$. Clearly, $\vec{c}$ is $h$-feasible (according to Definition 2) if and only if all elements of $\vec{c}$ are integers and $\vec{c} \prec \vec{c}_h$. From the perspective of majorization, we now reexamine some of our results and make connections with existing related works.

First, Theorem 1 states that $T(h)$ is ancestral $\vec{c}$-colorable if and only if $\vec{c} \prec \vec{c}_h$. Note that it is trivial that $T(h)$ is ancestral $\vec{c}_h$-colorable (one color for each layer). Equivalently, $T'(h)$, which is obtained from $T(h)$ by adding edges between each node and all of its descendants, is $\vec{c}_h$-colorable (i.e., it can be colored using $c_i$ Color $i$ so that adjacent vertices have different colors). A result similar to Theorem 1 was proved by Folkman and Fulkerson [19] but for the edge coloring problem for a general graph, which states that if a graph $G$ is $\vec{c}$-colorable and $\vec{c}' \prec \vec{c}$ then $G$ is also $\vec{c}'$-colorable. However, the technique used in [19] for edge coloring doesn’t work in the context of vertex coloring, which turns out to be much more complicated. The same conclusion for vertex coloring, as far as we know, only holds for claw-free graphs (see de Werra [18], and also [17, Theorem 4]). Recall that a claw-free graph is a graph that doesn’t have $K_{1,3}$ as an induced subgraph. However, our graph $T'(h)$ contains a claw when $h \geq 3$, e.g., the subgraph induced by Nodes 2, 4, 10, and 11 (see Fig. 3). Therefore, this result doesn’t cover our case.

Second, in the language of majorization, the Color-Splitting Algorithm starts from an integer vector $\vec{c}$ of dimension $h$ that is majorized by $\vec{c}_h$ and creates two new integer vectors, both of dimension $h - 1$ and majorized by $\vec{c}_{h-1}$. Applying the algorithm recursively produces $2^{h-1}$ sequences of vectors (of decreasing dimensions) where the $i$th element in each sequence is majorized by $\vec{c}_{h-i+1}$.

In the remainder of this section, we discuss a potential way to solve the ancestral tree coloring problem using a geometric approach. It was proved by Hardy, Littlewood, and Pólya in 1929 that $\vec{x} \prec \vec{y}$ if and only if $\vec{x} = \vec{y}P$ for some doubly stochastic matrix $P$. Moreover, Birkhoff theorem ensures that every doubly stochastic matrix can be expressed as a convex combination of permutation matrices. With these, a geometric interpretation of majorization was established that $\vec{x} \prec \vec{y}$ if and only if $\vec{x}$ lies in the convex hull of all the permutations of $\vec{y}$ [21]. Therefore, it is natural to consider a geometric approach to the ancestral coloring problem. The following example is one such attempt.
Example 4. Let \( h = 3 \). Recall that \( \vec{c}_h = [2, 4, 8] \). Also, let \( \vec{c} \rightarrow [4, 5, 5] \) and \( \vec{c}'' \rightarrow [3, 4, 7] \). Clearly, \( \vec{c} \prec \vec{c}_h \) and \( \vec{c}'' \prec \vec{c}_h \), and hence the CSA would work for both cases. Alternatively, we first write \( \vec{c} \) and \( \vec{c}'' \) as convex combinations of permutations of \( \vec{c}_h \) as

\[
\vec{c} = \frac{1}{2}[4, 2, 8] + \frac{1}{2}[4, 8, 2],
\]

(7)

\[
\vec{c}'' = \frac{1}{4}[2, 4, 8] + \frac{1}{4}[2, 8, 4] + \frac{1}{2}[4, 2, 8].
\]

(8)

Note that every permutation of \( \vec{c}_h \) admits a trivial layer-based ancestral coloring as shown in the two trees at the top in Fig. 7. Now, (7) demonstrates that it is possible to obtain an ancestral coloring for \( \vec{c} \) by taking a half of the tree colored by \([4, 2, 8]\) and a half of that colored by \([4, 8, 2]\) and stitching them together, which is precisely what we illustrate in Fig. 7. Similarly, Fig. 8 shows how we obtain an ancestral coloring for \( \vec{c}'' \) by taking a quarter of the tree colored by \([2, 4, 8]\), a quarter of the tree colored by \([2, 8, 4]\), and a half of the tree colored by \([4, 2, 8]\) and stitching them together.

Fig. 8. A \([3R, 4G, 7B]\) ancestral coloring for \( T_3 \) can be obtained by “stitching” a quarter of the coloring \([2R, 4G, 8B]\), a quarter of the coloring \([2R, 8G, 4B]\), and a half of the coloring \([4R, 2G, 8B]\). Note that \([3, 4, 7] = \frac{1}{4}[2, 4, 8] + \frac{1}{2}[2, 8, 4] + \frac{1}{2}[4, 2, 8]\).

With the success of this approach for the above examples \((h = 3)\) and other examples with \( h = 4 \) (not included for the sake of conciseness), it is tempting to believe that for every \( \vec{c} \) majorized by \( \vec{c}_h \), an ancestral coloring can be obtained by stitching sub-trees of trivial layer-based ancestral-colored trees. However, our initial attempt for \( h = 5 \) indicates that it could be quite challenging to obtain ancestral colorings by such a method, even if a convex and dyadic combination can be identified. Further investigation on this direction is left for future work.

III. Efficient Parallel Private Retrieval of Merkle Proofs via Balanced Ancestral Colorings

A. Problem Description and A Proposed Solution

1) Merkle Tree and Merkle Proof: We first provide some relevant background. A Merkle tree is a binary tree in which each node is the (cryptographic) hash of the concatenation of the contents of its child nodes [12]. A Merkle tree represents a large amount of data in a way that not only guarantees
the integrity of data (any change in the content will lead to the change in the root of the tree) but also allows very efficient membership testing, which can be performed with complexity $\log(n)$ where $n$ is the number of data items stored at the leave nodes. Because of its simple construction and powerful features, Merkle trees have been widely used in practice, e.g., for data synchronization in Amazon DynamoDB [6], for certificates storage in Google’s Certificate Transparency [22], and for transactions storage in blockchains [1, 23], among many other applications.

Fig. 9. Illustration of a Merkle tree that stores eight transactions $T_1, T_2, ..., T_8$. The leaf nodes of the tree store the hashes of these transactions, i.e., $H_1 = H(T_1), ..., H_8 = H(T_8)$, where $H(\cdot)$ is a cryptographic hash function. Each non-leaf node is created by hashing the concatenation of the hashes stored at its two child nodes. For instance, $H_{12} = H(H_1 || H_2)$, and $H_{1234} = H(H_{12} || H_{34})$. The Merkle proof of the transaction $T_3$ consists of $H_1, H_{12}, H_{5678}$, and $H_{18}$, that is, all the hashes stored at the siblings of nodes along the path from $H_1$ up to the root.

In Fig. 9 we have a Merkle tree storing eight transactions $(T_i)_{i=1}^8$ at the bottom. The leaf nodes store the hashes $(H_i)_{i=1}^8$ of these transactions, where $H_i = H(T_i)$, $1 \leq i \leq 8$, and $H(\cdot)$ is a cryptographic hash, e.g. SHA256 [24]. Because a cryptographic hash function is collision resistant, i.e., given $H_X = H(X)$, it is computationally hard to find $Y \neq X$ satisfying $H(Y) = H(X)$, no change in the underlying set of transactions can be made without changing the Merkle root. Therefore, once the Merkle root is published, no one can modify any transaction while keeping the same root hash. That is why Merkle tree is used to prevent tempering to data. On top of that, the binary tree structure allows an efficient membership/inclusion test: an user with a transaction, e.g., $T_3$, can verify that this transaction is indeed included in the Merkle tree with the published root, e.g., $H_{18}$, by downloading the corresponding Merkle proof, which includes $H_1, H_{12}, H_{5678}$, and $H_{18}$. The root hash, which has been published before, usually doesn’t need to be downloaded. In general, a Merkle proof consists of $h = \lceil \log(n) \rceil$ hashes, where $h$ is the height and $n$ the number of leaves. In this example, the user can compute $H_{34} = H(H_3 || H_4)$, and then $H_{1234} = H(H_{12} || H_{34})$, and finally verify that $H_{18} = H(H_{1234} || H_{5678})$. The membership test is successful if the last equality holds.

Next, we give a brief overview of the literature on the private information retrieval problem and then discuss our problem of private retrieval of Merkle proofs as well as the high-level description of the proposed solution.

2) Private Information Retrieval: The Private Information Retrieval (PIR) problem was introduced by Chor et al [25], which raised the following question: is it possible for a user to retrieve an item in a database without letting the server(s) know which item is being retrieved? The answer given in [25] is Yes, and the trick is to replicating the database over several servers. For example, if the database is $X = \{x_i\}_{i=1}^n$, which is stored at two servers, and the user wishes to retrieve $x_j$ privately, then it can choose an arbitrary set $I \subseteq \{1, 2, \ldots, n\}$ and requests the XOR-sum $\sum_{i \in I} x_i$ from the first server and the XOR-sum $x_j + \sum_{i \in I} x_i$ from the second server. If the two servers are not colluding, then each does not know anything about which item is being retrieved. This kind of PIR scheme is
categorized as an information-theoretic scheme, referred to as IT-PIR, which requires multiple non-colluding servers. Later, Kushilevitz et al. [26] developed the second category of PIR schemes called computational PIR or cPIR for short, which relies on computationally hard problems and requires a single server. Since then, a great number of different PIR schemes in both categories have been developed and PIR still remains a very active research area in the field of computer science. The latest practical developments (theory + implementation) include SEALPIR [27], [14], which is based on Microsoft SEAL (Simple Encrypted Arithmetic Library) [28], and MulPIR of Google [29].

3) Private Retrieval of Merkle Proofs: The problem is defined as follows: Suppose that \( n \) items \((\mathcal{T}_i)_{i=1}^n\) are stored in a Merkle tree of height \( h \) and that the Merkle root (root hash) is published and known to everyone; design a scheme that allows a user who owns one item to download a Merkle proof of that item (in order to verify if the item indeed belongs to that Merkle tree) without revealing the item to the server(s) that store(s) the Merkle tree. This problem has a direct application in all systems that use Merkle tree to store data, in particular, blockchains, in which privacy has always been a pressing issue ([8], [9], [10]). An efficient solution to this problem allows a client who has a transaction (or received from someone else) to privately verify whether the transaction belongs to a certain block or not by privately downloading the Merkle proof of the transaction from a full node storing the corresponding block. In the context of website certificates [2], such a solution allows a web client to verify with a log server whether the certificate of a website is indeed valid (i.e., included in the log) without revealing which website it is visiting (see [11] for a more detailed discussion). We note that Lueks and Goldberg [11] investigated the same problem and addressed it by developing an IT-PIR solution, which requires multiple (log) servers and works under the assumption that the servers do not collude. Our solution is based on a cPIR and requires a single server only, thus avoiding the risk of server collusion.

At the first glance, by treating the set of hashes in the tree as a database, this problem seems to be a straightforward batch version of the PIR problem, in which a subset of items instead of a single one needs to be retrieved. Note that a Merkle proof consists of \( h \) hashes and hence the batch size is \( h \). There have been quite a few works in the literature dealing with batch PIR in both information-theoretic and computational settings, see, e.g., [30], [31], [11], [32], and references therein. However, a careful examination of the problem reveals two unique aspects to this problem compared to a general batch PIR: first, the basic assumption in PIR that the index of the item in the database is known is unrealistic (practical schemes like SealPIR introduce complicated methods to tackle this issue, e.g., by introducing an additional Oracle, which has its own issue), and second, a Merkle proof does not contain an arbitrary set of \( h \) hashes like in the batch PIR’s setting; indeed, there are only \( n = 2^h \) such proofs compared to \( \binom{2^n}{h-2} \) subsets of random \( h \) hashes in the Merkle tree. The reduction in the number of possible sets to be retrieved should be and could be exploited to speed up the whole process, as shown in our detailed discussion in Section III-C. In short, our parallel implementation of SealPIR ([27]) using balanced ancestral colorings of Merkle tree developed in this work can improve the computation speed compared to the baseline by a factor of \( h/2 \).

Per our aforementioned discussion, we propose to solve the Private Retrieval of Merkle Proofs problem in two stages. In the first stage (see Section III-B), we deal with the problem of Private Index Retrieval, in which a user has an item that belongs to an ordered database of \( n \) items, stored by a Merkle tree, and wants to determine the index of that item in the database. In the second stage (see Section III-C), once the index of the item in the bottom layer of the Merkle tree has been found, we use a balanced ancestral coloring of the tree to speed up the parallel implementation of SealPIR to retrieve the corresponding Merkle proof without revealing the item to the server storing the tree.

B. Private Index Retrieval Problem

The Private Index Retrieval (PIXR) Problem is stated as follows. A server stores a list of \( n \) distinct \( k \)-bit vectors \( v_1, \ldots, v_n \). A client possesses a vector \( v \), which is among the \( n \) vectors. The client wishes to know the index \( i \in \{1, 2, \ldots, n\} \) so that \( v = v_i \). Suppose that \( n \) and \( k \) are known a priori, what is most efficient method (in terms of computation and communication costs) for the client to extract this index without revealing \( v \) to the server? In the context of Merkle trees, because SHA256
is the most widely used hash function, we usually have \( k = 256 \). Moreover, we experiment with \( n = 2^h \) where \( 10 \leq h \leq 23 \). Note that in our experiments, instead of using blockchain transactions, we use their 256-bit hashes as \( v_i \), which are at least 8 times smaller than the transactions themselves.

A trivial approach to determine the index \( i \) is for the client to download the entire list of \( n \) vectors and compare one by one against \( v \) to find \( i \) satisfying \( v = v_i \). This approach requires no computation at the server side but a computation complexity of \( O(n) \) at the client side and a communication cost of \( O(kn) \) bits.

To reduce the communication cost, we observe that when \( k \) is much greater than \( \log_2(n) \), we usually do not need all \( k \) bits to distinguish the \( n \) vectors. As we often deal with \( k = 256 \) and \( n = 2^h \) where \( h \ll 256 \), this observation is relevant. If only communication cost is our concern, then this is precisely the Distinct Vectors Problem, which was first discussed in [33]; the goal is to find a minimum set of bits among \( k \) bits that can still distinguish all \( n \) vectors. It was shown in [33] that the problem is NP-hard. However, in the context of our work, where the vectors are 256-bit hashes and can be effectively treated as randomly distributed vectors, \( 2\log_2 n \) bits are usually enough to distinguish \( n \) vectors (as contrast to the worst case where all \( k \) bits may be required to distinguish the vectors). Indeed, the probability that \( n \) random \( (2\log_2 n) \)-bit vectors are pairwise distinct is \( p(n) = \prod_{i=1}^{n} \frac{n^2-i}{n^2} \), which is sufficiently large. The minimum number of bits often lies between \( \log_2 n \) and \( 2\log_2 n \), and so, \( 2\log_2 n \) provides a sufficiently good approximation. This observation has also been confirmed through extensive experiments on several sets of hashes of real Bitcoin transactions. Note that the proposed method requires some extra computation at the server side (performed once) and the same computation cost (if not faster) at the client side as in the trivial method. Moreover, the communication cost is reduced by a factor of \( k/2\log_2 n \), which ranges from 4x to 12x for \( n = 2^h \), \( 10 \leq h \leq 30 \) when \( k = 256 \).

| Number of transactions | Vector length | Average run time | Average number of steps |
|------------------------|--------------|------------------|------------------------|
| \( 2^{10} \)          | 20 bits      | 7 ms             | 1.6                    |
| \( 2^{12} \)          | 24 bits      | 16 ms            | 2.3                    |
| \( 2^{14} \)          | 28 bits      | 55 ms            | 1.7                    |
| \( 2^{16} \)          | 32 bits      | 226 ms           | 1.3                    |
| \( 2^{18} \)          | 36 bits      | 705 ms           | 1.4                    |
| \( 2^{19} \)          | 38 bits      | 1700 ms          | 1.7                    |
| \( 2^{20} \)          | 40 bits      | 3668 ms          | 1.5                    |
| \( 2^{21} \)          | 42 bits      | 8929 ms          | 2.1                    |
| \( 2^{22} \)          | 44 bits      | 25253 ms         | 1.7                    |
| \( 2^{23} \)          | 46 bits      | 56750 ms         | 1.3                    |

**Table I**

The running time of the algorithm at the server side that finds a set of \( 2\log_2 n \) bits that can distinguish \( n \) 256-bit hashes. These hashes are from real transactions collected from blockchain.com API. The measured running times were averaged over 10 datasets of each size and confirm the theory that only \( 2\log_2 n \) bits are usually required to distinguish \( n \) 256-bit hashes.

We ran a simple algorithm that extracts \( 2\log_2 n \) bits that distinguish all \( n \) hashes of transactions downloaded from blockchain.com API [34]. The algorithm checks whether the set of the first \( 2\log_2 n \) bits can be used to distinguish all the hashes (using a hash table). If Yes, then it returns these bit positions (providing the position of the first bit is enough as they are consecutive), if No, then it jumps to the next set of \( 2\log_2 n \) bits and repeats the test, and so on. For each size \( n \), we ran our experiment on 10 different datasets and took the average measurement. As shown in Table I the algorithm only needs to check 1 to 2 sets of \( 2\log_2 n \) bits before obtaining a relevant bit sets. Moreover, the running times are relatively low. Note that this is a one-off computation for each Merkle tree at the server.

**C. An Efficient Parallel Implementation of SealPIR Using Balanced Ancestral Coloring of Merkle Trees**

Suppose that the client has a transaction and knows its index in the Merkle tree (see Section III-B). Note that the index retrieval phase does not rule out the case in which the transaction does not belong
to the block. Now, the client wants to privately and efficiently download the Merkle proof of that transaction and verify if the transaction is really included in the block. Given the index, the client knows exactly which hashes to download from the Merkle tree. A trivial method is to download the whole tree. This does not incur extra computation but requires a large bandwidth. The second option (proofs-as-elements) is to run an efficient computational PIR scheme, e.g., SealPIR [13], [14] on a new set of \( n = 2^h \) elements (\( h \) is the height of the tree), each of which corresponds to a Merkle proof and therefore has size \( 256 \times h \) bits. This seemingly natural solution turns out to be the most expensive one because it creates a database \( h/2 \) times larger than the Merkle tree itself. The third option (layer-based) is to run the same PIR scheme in parallel on different layers to privately retrieve one hash from each. The computation time is dominated by the PIR scheme running on the bottom layer, which contains \( n = 2^h \) hashes.

![Fig. 10. Illustration of a (colored) swapped Merkle tree, in which sibling nodes, e.g., \( H_{1234} \) and \( H_{5678} \), have been swapped. A Merkle proof in the ordinary Merkle tree (see Fig. 9) corresponds to a root-to-leaf path in the swapped Merkle tree. Thanks to the Ancestral Property, each root-to-leaf path (ignoring the root) contains exactly one node for each color, which implies that PIR schemes can be simultaneously applied to different color classes without revealing the path.](image)

**Our proposal** is to use a balanced ancestral coloring of \( T(h) \), which is obtained from a Merkle tree by swapping its sibling nodes (see Fig. 10), to partition the tree nodes into (almost) equal-sized parts, and then run a PIR scheme on each part. As sibling nodes have been swapped, the set of nodes forming a Merkle proof (excluding the Merkle root) in the ordinary Merkle tree corresponds to a set of nodes along a root-to-leaf path (ignoring the root) in \( T(h) \). The Ancestral Property of the coloring guarantees that these nodes have different colors, i.e., belong to different parts, which implies that each part (corresponding to a color class) is queried exactly once (with a PIR query). Therefore, since each PIR scheme can retrieve a node without revealing its identity, all paths are indistinguishable. Thus, the server cannot figure out which path (i.e., which Merkle proof) is being retrieved.

**Experimental setup.** We ran our experiments using Oracle instances in Melbourne, Australian domain. We ran the SealPIR [14] server and client in parallel on multiple cores on a VM.Standard2.24 (Intel processors, Virtual machine, 24 core OCPU, 320 GB memory, 24.6 Gbps network bandwidth, Block storage only), all running Oracle Linux 8.4-2021.10.04-0. We compiled our C++ code with Cmake version 3.22 and Microsoft SEAL version 3.2.0 [28]. Finally, the total running times (communication and computation) of four different solutions are compared assuming a typical bandwidth of 100 Mbps. In our proposed solution, we assumed that a balanced ancestral coloring already exists and excluded the time spent to find one, which incurs a one-time cost of a few minutes at most.

The C++ code of the SealPIR library was designed to execute only on one core. To serve our purpose, we created multiple threads to run on multiple cores. In the parallel mode, each thread ran a PIR scheme on a different database on a separate core in our Oracle instance. The size of each
database varied, depending on the scheme (layer-based or coloring-based). For example, if \( h = 10 \), then we will need ten threads, each of which will run on a different database with size ranging from \( 2^1 \) to \( 2^{10} \) for the layer-based scheme. For the coloring-based scheme, all databases have roughly the same size \( (2^{11} - 2)/10 \). The percentages of the computation time in our coloring-based scheme are reported in Table II. Note that for bandwidths larger than 100Mbps, the computation time will contribute to a higher percentage of the total running time. Also, a larger bandwidth will make the trivial algorithm faster, and as a consequence, the threshold of the data size \( (n) \) where PIR-based scheme like ours can beat the trivial solution will be increased.

![Comparison of total running times](image)

Fig. 11. A comparison of the total running times (communication and computation) of four different solutions (trivial, proofs-as-elements, layer-based, and coloring-based solutions) in the network bandwidth 100 Mbps. Our coloring-based solution always beats the layer-based solution, and from \( n = 2^{17} \), it also beats the trivial solution.

| Database size \( (n) \) | \( 2^1 \) | \( 2^2 \) | \( 2^3 \) | \( 2^4 \) | \( 2^5 \) | \( 2^6 \) |
|-------------------------|---|---|---|---|---|---|
| Computation / (Computation + Communication) (%) | 5.22 | 5.06 | 5.85 | 6.86 | 7.84 | 8.98 | 9.06 |

| Database size \( (n) \) | \( 2^1 \) | \( 2^2 \) | \( 2^3 \) | \( 2^4 \) | \( 2^5 \) | \( 2^6 \) |
|-------------------------|---|---|---|---|---|---|
| Computation / (Computation + Communication) (%) | 11.76 | 16.31 | 18.93 | 26.33 | 33.92 | 43.25 | 55.20 |

TABLE II
THE PERCENTAGE OF THE COMPUTATION TIME (SERVER + CLIENT) WITH RESPECT TO THE WHOLE RUNNING TIME (COMPUTATION + COMMUNICATION) IN OUR COLORING-BASED SCHEME ASSUMING THE NETWORK BANDWIDTH BETWEEN THE SERVER AND CLIENT IS 100 Mbps.

IV. Conclusions

We introduce in this work a novel coloring problem for rooted trees, referred to as ancestral coloring, which has an immediate application in private retrieval of Merkle proofs in a Merkle tree. More specifically, a balanced ancestral coloring of a Merkle tree of height \( h \) allows us to speed up the retrieval process by a factor of \( \Theta(h/2) \) compared to a straightforward parallel version of any private information retrieval scheme. The ancestral coloring problem has strong connections to other major topics in discrete mathematics including vertex and edge colorings of graphs using a given number of each color, combinatorial batch codes, and (integer) majorizations.

We settle the problem completely for perfect binary trees by establishing a necessary and sufficient condition to efficiently identify all color sequences \( \vec{c} \), where \( c_i \) refers to the number of nodes to have Color \( i \), that are feasible for a perfect binary tree of height \( h \). In particular, we propose a divide-and-conquer algorithm called Color-Splitting Algorithm that can color the tree with any given feasible color sequence \( \vec{c} \) in almost linear time. The algorithm, implemented in Java [7], can return a valid
coloring of a tree of two billions nodes in less than five minutes. Furthermore, we implement a parallel scheme on top of SealPIR, a state-of-the-art private information retrieval scheme, to retrieve Merkle proofs based on balanced ancestral colorings of Merkle trees. Our implementation outperforms a straightforward parallel version of SealPIR, which verifies our theoretical claim.

We discuss below several intriguing open problems for future research.

**Open Problem 1.** An immediate open problem is to find characterizations of feasible (ancestral) color sequences for perfect $q$-ary trees with $q \geq 2$ (e.g., $q = 128$ as in the case of Swarm [35], the distributed storage system underlying Ethereum) and more generally, for arbitrary rooted trees. While the former seems manageable, the latter could be quite challenging.

![Fig. 12. A rooted tree $T$ of height $h = 3$ with ancestral balanced index $b(T) = 2$. There is no balanced ancestral coloring with $h = 3$ colors for this tree. However, there exists a balanced one if we add one more color, i.e., $h^*(T) = 4$.](image)

**Open Problem 2.** We observe that not every rooted tree (even restricted to the binary ones) has a balanced ancestral coloring (using $h$ colors). For example, the tree in Fig. [12] has nine nodes (except the root) and if we color the left child of the root with Red, then to satisfy the Ancestral Property there can be no more than two red nodes, which implies that a balanced ancestral coloring using $h = 3$ colors does not exist for this tree. Let $b_C(T) \triangleq \max_{1 \leq i, j \leq h}(|C_i| - |C_j|)$ denote the balance index of an ancestral coloring $C = \{C_1, C_2, \ldots, C_h\}$ of a rooted tree $T$. A balanced ancestral coloring has balanced index 0 or 1. Given a rooted tree $T$, an intriguing question is how to find the most balanced ancestral coloring with $h$ colors for the tree, i.e., the one with the smallest balance index. In other words, given a rooted tree $T$, one could be interested in finding an efficient algorithm that computes its ancestral balance index, defined as $b(T) \triangleq \min_C b_C(T)$, where the minimum is taken over all $h$-color ancestral colorings $C$ of $T$. From the application perspective, a more balanced coloring leads to a faster retrieval scheme.

**Open Problem 3.** Given an arbitrary rooted tree of height $h$, what is the smallest number of colors $h^* \geq h$ required to have a balanced ancestral coloring for the tree? For example, the tree in Fig. [12] requires at least $h^* = 4$ colors to have a balanced ancestral coloring. Note that a trivial balanced ancestral coloring of a tree assigns every node a distinct color. From the application perspective, the number of colors required is the same as the number of CPU cores needed to run the parallel retrieval scheme, which means that the smaller the number of colors the better.

**Open Problem 4.** Classify/enumerate all ($h$-color) balanced ancestral colorings of a perfect binary trees of height $h$. Note that the Color-Splitting Algorithm only generates one of such colors.

V. ACKNOWLEDGMENTS

This work was supported by the Australian Research Council through the Discovery Project under Grant DP200100731 and carried out on Oracle virtual machines, supported by Oracle for Research.
APPENDIX A

PROOF OF LEMMA 6

Before proving Lemma 6, we need an auxiliary result.

Lemma 7. Let $h \geq 4$ and $\vec{c}, \vec{a},$ and $\vec{b}$ be defined as in Lemma 3. As $\vec{c}$ is a sorted sequence, we can partition its elements from $c_4$ to $c_h$ into $k$ different runs where within each run all $c_i$’s are equal,

$$c_4 = \cdots = c_i < c_{i+1} = \cdots = c_k < \cdots < c_{k-1} = \cdots = c_h.\$$

For any $1 \leq r \leq k$, let $A_r \triangleq \sum_{j=4}^{i_r} a_j$, $B_r \triangleq \sum_{j=4}^{i_r} b_j$, and $C_r \triangleq \sum_{j=4}^{i_r} c_j$. Then $\min\{A_r, B_r\} \geq \lceil \frac{C_r}{2} \rceil - 1$.

Proof. First, we have $|(a_2 + a_3) - (b_2 + b_3)| = |(c_2 + c_3 - a_2) - (c_2 + c_3 - b_2)| \leq 1$. Moreover, the way that $a_i$ and $b_i$ are defined for $i \geq 4$ in Lemma 3 guarantees that

$$|(a_2 + a_3 + A_r) - (b_2 + b_3 + B_r)| \leq 1,$$

which implies that $|A_r - B_r| \leq 2$. Furthermore, as $a_i + b_i = c_i$ for all $i \geq 4$, we have

$$A_r + B_r = \sum_{j=4}^{i_r} (a_j + b_j) = \sum_{j=4}^{i_r} c_j = C_r.$$

Thus, $\min\{A_r, B_r\} \geq \lceil \frac{C_r}{2} \rceil - 1$. \hfill \Box

Proof of Lemma 6. We use a similar approach to that of Lemma 5. However, the proof is more involved because we now must take into account the relative positions of $a_2, a_3, b_2,$ and $b_3$ within each sequence after being sorted, and must treat $\vec{a}$ and $\vec{b}$ separately. The case with $h = 2$ or 3 is straightforward to verify. We assume $h \geq 4$ for the rest of the proof.

Let $\vec{a}' = [a'_2, \ldots, a'_h]$ and $\vec{b}' = [b'_2, \ldots, b'_h]$ be obtained from $\vec{a}$ and $\vec{b}$ after sorting. According to Definition 2, the goal is to show that (C1) and (C2) hold for these two sequences while replacing $h$ by $h - 1$. The proof that (C2) holds for $\vec{a}'$ and $\vec{b}'$ is identical to that of Lemma 5, implied by the following facts: first, $\sum_{i=2}^{h} a'_i = \sum_{i=2}^{h} b'_i = \sum_{i=2}^{h} b_i$, and second, due to the definitions of $a_i$ and $b_i$, the two sums $\sum_{i=2}^{h} a_i$ and $\sum_{i=2}^{h} b_i$ are exactly the same and equal to $\frac{1}{2}(\sum_{j=1}^{h-1} c_j - 2) = \frac{1}{2}\sum_{i=2}^{h-1} 2^i = \sum_{i=1}^{h-1} 2^i$. It remains to show that (C1) holds for these color sequences.

As $\vec{c}$ is a sorted sequence, we partition its elements from $c_4$ to $c_h$ into $k$ different runs where within each run all $c_i$’s are equal,

$$c_4 = \cdots = c_i < c_{i+1} = \cdots = c_k < \cdots < c_{k-1} = \cdots = c_h.\tag{9}$$

For $r = 1, 2, \ldots, k$, define the $r$-th run as $R_r \triangleq [i_{r-1} + 1, i_r]$, where $i_0 \triangleq 3$. Then (9) means that for each $r \in [k]$, $c_j$’s are the same for all $j \in R_r$ and moreover, $c_j < c_{j'}$ if $j \in R_r$, $j' \in R_{r'}$, and $r < r'$. As $\vec{c}$ is $h$-feasible, it satisfies (C1),

$$c_1 + c_2 + c_3 + \sum_{j=4}^{i_r} c_j + \cdots + \sum_{j=i_{r-1}+1}^{i_r} c_j = \sum_{j=1}^{i_r} c_j \geq \sum_{j=1}^{i_r} 2^j.$$

Equivalently, setting $C_r \triangleq \sum_{j=4}^{i_r} c_j + \cdots + \sum_{j=i_{r-1}+1}^{i_r} c_j$, we have

$$c_1 + c_2 + c_3 + C_r \geq \sum_{j=1}^{i_r} 2^j.\tag{10}$$

We will make extensive use of this inequality later.

In order to show that (C1) holds for $\vec{a}'$ and $\vec{b}'$, our strategy is to first prove that (C1) holds for these sequences at the end-points $\ell = i_r$ of the runs $R_r$, $r \in [k]$, and then employ Lemma 4 to conclude that (C1) also holds for these sequences at all the middle-points of the runs. We also need to demonstrate that (C1) holds for $\vec{a}'$ and $\vec{b}'$ at the indices of $a_2, a_3, b_2,$ and $b_3$ within these sorted sequences. Notice the differences with the proof of Lemma 5, first, we consider the runs from $c_4$
instead of \( c_2 \), and second, we must take into account the positions of \( a_2, a_3, b_2, b_3 \) relative to the runs within the sorted sequence \( \vec{a}' \) and \( \vec{b}' \).

Since \( |c_i/2| \leq a_i \leq |c_i/2| \) for every \( 2 \leq i \leq \ell \), it is clear that \( a_i \leq a_i' \) if \( i \in R_r, i' \in R_{r'}, \) and \( r < r' \). Therefore, \( \vec{a}' \) can be obtained from \( \vec{a} \) by sorting its elements \( a_4, \ldots, a_h \) locally within each run and then inserting \( a_2 \) and \( a_3 \) into their correct positions (to make \( \vec{a}' \) non-decreasing). The same conclusion holds for \( \vec{b}' \). Note also that \( a_2 \) and \( a_3 \) are inserted between runs (unless they are the first or the last element in \( \vec{a}' \)) and do not belong to any run. The same statement holds for \( b_2 \) and \( b_3 \).

**First, we show that \( \vec{a}' \) satisfies (C1).** We divide the proof into two cases depending on whether \( a_2 \leq a_3 \) or not.

- (Case a1) \( a_2 \leq a_3 \). The sorted sequence \( \vec{a}' \) has the following format in which within each run \( R_r \), \( j \in R_r \), ordered so that those \( a_j = \left[ \frac{c_j}{2} \right] \) precede those \( a_j = \left[ \frac{c_j}{2} \right] \). Note that \( c_j \)'s are equal for all \( j \in R_r \).

\[
\begin{array}{ccc}
\text{runs} & a_2 & \text{runs} & a_3 & \text{runs}
\end{array}
\]

Note that it is possible that there are no runs before \( a_2 \), or between \( a_2 \) and \( a_3 \), or after \( a_3 \).

*In the first sub-case*, the index of interest \( \ell = i_r \) is smaller than the index of \( a_2 \) in \( \vec{a}' \). In order to show that (C1) holds for \( \vec{a}' \) at \( \ell = i_r \), we prove that

\[
A_r \triangleq \sum_{j=4}^{i_r} a_j \geq \sum_{j=1}^{i_r-3} 2^j,
\]

noting that in this sub-case the set \( \{a_j : 4 \leq j \leq i_r\} \) corresponds precisely to the set of the first \( i_r - 3 \) elements in \( \vec{a}' \). We demonstrate below that (11) can be implied from (10). Indeed, since \( \vec{c} \) is non-decreasing, from (10) we deduce that

\[
4C_r \geq c_1 + c_2 + c_3 + C_r \geq \sum_{j=1}^{i_r} 2^j.
\]

Combining this with Lemma 7 we obtain the desired inequality (11) as follows.

\[
A_r \geq \left[ \frac{C_r}{2} \right] - 1 \geq \left[ \frac{1}{8} \sum_{j=1}^{i_r} 2^j \right] - 1 = \left[ \sum_{j=1}^{i_r-3} 2^j + \frac{14}{8} \right] - 1 = \sum_{j=1}^{i_r-3} 2^j.
\]

*In the second sub-case*, the index of interest \( \ell \) is greater than or equal to the index of \( a_2 \) but smaller than that of \( a_3 \) in \( \vec{a}' \). In other words, either \( \ell = i_r \) is greater than the index of \( a_2 \) or \( \ell \) is precisely the index of \( a_2 \) in \( \vec{a}' \). If the latter occurs, let \( i_r \) be the end-point of the run preceding \( a_2 \) in \( \vec{a}' \). To prove that (C1) holds for \( \vec{a}' \) at \( \ell \), in both cases we aim to show that

\[
a_2 + A_r \triangleq a_2 + \sum_{j=4}^{i_r} a_j \geq \sum_{j=1}^{i_r-2} 2^j,
\]

noting that \( \{a_2\} \cup \{a_j : 4 \leq j \leq i_r\} \) forms the set of the first \( i_r - 2 \) elements in \( \vec{a}' \). We demonstrate below that (12) is implied by (10). **First**, since \( \vec{c} \) is non-decreasing, from (10) we deduce that

\[
(c_1 + c_2) + 2C_r \geq (c_1 + c_2) + (c_3 + C_r) \geq \sum_{j=1}^{i_r} 2^j,
\]

which implies that

\[
\frac{c_1 + c_2}{4} + \frac{C_r}{2} \geq \frac{1}{4} \sum_{j=1}^{i_r} 2^j = \sum_{j=1}^{i_r-2} 2^j + \frac{3}{2}.
\]

Next, since \( c_2 \geq c_1 \geq 3 \), we have

\[
a_2 = c_2 - 1 > \frac{c_2}{2} = \frac{2c_2}{4} \geq \frac{c_1 + c_2}{4}.
\]
Then, by combining this with Lemma 7 and (13), we obtain the following inequality
\[ a_2 + A_r > \frac{c_1 + c_2}{4} + \left( \left\lfloor \frac{C_r}{2} \right\rfloor - 1 \right) \geq \frac{c_1 + c_2}{4} + \frac{C_r - 1}{2} \cdot \frac{3}{2} \geq \sum_{j=1}^{i_r-2} 2^j. \]
Thus, (12) follows.

In the third sub-case, the index of interest \( \ell \) is greater than or equal to the index of \( a_3 \). In other words, either \( \ell = i_r \) is greater than the index of \( a_3 \) or \( \ell \) is precisely the index of \( a_3 \) in \( a' \). If the latter occurs, let \( i_r \) be the end-point of the run preceding \( a_3 \) in \( a' \). To prove that (C1) holds for \( a' \) at \( \ell \), in both cases we aim to show that
\[ a_2 + a_3 + \sum_{j=1}^{i_r} a_j \geq \sum_{j=1}^{i_r-2} 2^j, \quad \text{(14)} \]
noting that in this sub-case \( \{a_2, a_3\} \cup \{a_j : 4 \leq j \leq i_r\} \) forms the set of the first \( i_r - 1 \) elements in \( a' \). This turns out to be the easiest sub-case. With the presence of both \( a_2 \) and \( a_3 \) in the sum on the left-hand side of (14), according to the way \( a_i \) and \( b_i \) are selected in Lemma 6, we have
\[ \left| \left( a_2 + a_3 + \sum_{j=1}^{i_r} a_j \right) - \left( b_2 + b_3 + \sum_{j=4}^{i_r} b_j \right) \right| \leq 1, \]
and since \( a_2 + a_3 + b_2 + b_3 = c_1 + c_2 + c_3 - 2 \) and \( a_j + b_j = c_j \) for \( j \geq 4 \), we also have
\[ a_2 + a_3 + \sum_{j=4}^{i_r} a_j + \left( b_2 + b_3 + \sum_{j=4}^{i_r} b_j \right) = \sum_{j=1}^{i_r} c_j - 2 \geq \sum_{j=2}^{i_r-2} 2^j = 2 \sum_{j=1}^{i_r-2} 2^j, \]
which, together, imply (14).

- **(Case a2) \( a_2 > a_3 \).** The sorted sequence \( a' \) has the following format.

\[
\begin{array}{cccc}
\text{runs} & a_3 & \text{runs} & a_2 & \text{runs}
\end{array}
\]

Note that it is possible that there are no runs before \( a_3 \), or between \( a_3 \) and \( a_2 \), or after \( a_2 \).

Again, due to Lemma 4, we only need to demonstrate that (C1) holds for \( a' \) at the end-point \( i_r \) of each run \( R_r \), \( 1 \leq r \leq k \), and at \( a_2 \) and \( a_3 \).

In the first sub-case, \( \ell = i_r \) is smaller than the index of \( a_3 \) in \( a' \). As \( a_2 \) and \( a_3 \) are not involved, the same proof as in the first-subcase of Case a1 applies.

In the second sub-case, the index of interest \( \ell \) is greater than or equal to the index of \( a_3 \) but smaller than that of \( a_2 \) in \( a' \). In other words, either \( \ell = i_r \) is greater than the index of \( a_3 \) and smaller than that of \( a_2 \), or \( \ell \) is precisely the index of \( a_3 \) in \( a' \). If the latter occurs, let \( i_r \) be the end-point of the run preceding \( a_3 \) in \( a' \). To prove that (C1) holds for \( a' \) at \( \ell \), we aim to show
\[ a_3 + A_r \geq a_3 + \sum_{j=4}^{i_r} a_j \geq \sum_{j=1}^{i_r-2} 2^j, \quad \text{(15)} \]
noting that in this sub-case \( \{a_3\} \cup \{a_j : 4 \leq j \leq i_r\} \) forms the set of the first \( i_r - 2 \) elements in \( a' \). We demonstrate below that (15) can be implied from (10) in both cases when \( i_r = 4 \) and \( i_r > 4 \). First, assume that \( i_r = 4 \), i.e., \( r = 1 \) and the run \( R_1 \) consists of only one index 4. Now, (10) can be written as
\[ c_1 + c_2 + c_3 + c_4 \geq 4 \sum_{j=1}^{4} 2^j = 30, \]
and what we need to prove is
\[ a_3 + \left\lfloor \frac{c_1}{2} \right\rfloor \geq 2 \sum_{j=1}^{2} 2^j = 6, \]
noting that the element in $\vec{a}'$ corresponding to $c_4$ is either $\lfloor c_4/2 \rfloor$ or $\lceil c_4/2 \rceil$. Equivalently, plugging in the formula for $a_3$, what we aim to show is
\[
\left\lceil \frac{c_3 + c_1 - c_2}{2} \right\rceil + \left\lfloor \frac{c_4}{2} \right\rfloor \geq 6. \tag{16}
\]
This inequality is correct because
\[
\left\lceil \frac{c_3 + c_1 - c_2}{2} \right\rceil + \left\lfloor \frac{c_4}{2} \right\rfloor \geq \left\lceil \frac{c_1}{2} \right\rceil + \left\lfloor \frac{c_4}{2} \right\rfloor \geq 6,
\]
where the first inequality holds because $c_3 \geq c_2$ and the second inequality holds because $c_1 \geq 3$ and $c_4 \geq 8$, given that $c_4 \geq c_3 \geq c_2 \geq c_1$ and $c_1 + c_2 + c_3 + c_4 \geq 30$. To complete this sub-case, we assume that $i_r > 4$. In this scenario, $C_r$ in (10) has at least two terms $c_j$’s, which are all greater than or equal to $c_2$ and $c_3$, and hence, $C_r \geq c_2 + c_3$. Therefore, using Lemma 7 we have
\[
a_3 + A_r = \left\lceil \frac{c_3 + c_1 - c_2}{2} \right\rceil + A_r \geq \left\lceil \frac{c_1}{2} \right\rceil + \left( \left\lceil \frac{C_r}{2} \right\rceil - 1 \right) > \frac{c_1}{4} + \left( \frac{C_r - 1}{2} - 1 \right) = \frac{1}{4}(c_1 + 2C_r) - \frac{3}{2} \geq \frac{1}{4}(c_1 + c_2 + c_3 + C_r) - \frac{3}{2} > \frac{1}{4} \sum_{j=1}^{i_r} 2^j - \frac{3}{2} = \sum_{j=1}^{i_r-2} 2^j.
\]
Thus, (15) follows.

In the third sub-case, assume that the index of interest $\ell$ is greater than or equal to the index of $a_2$. As both $a_2$ and $a_3$ are involved, the proof goes in exactly the same way as in the third sub-case of Case 1a. Thus, we have shown that $\vec{a}'$ also satisfies (C1) and is $(h - 1)$-feasible.

Next, we show that $\vec{b}$ satisfies (C1). The proof is very similar to that for $\vec{a}'$. We divide the proof into two cases depending on whether $b_2 \leq b_3$ or not.

- (Case b1) $b_2 \leq b_3$. The sorted sequence $\vec{b}$ has the following format in which within each run $R_r$ are the elements $b_j$, $j \in R_r$, ordered so that those $b_j = \lceil \frac{c_4}{2} \rceil$ precede those $b_j = \lfloor \frac{c_4}{2} \rfloor$. Note that $c_j$’s are equal for all $j \in R_r$.

| runs | $b_2$ | runs | $b_3$ | runs |
|------|------|------|------|------|

It is possible that there are no runs before $b_2$, or between $b_2$ and $b_3$, or after $b_3$.

The proof for the first and the third sub-cases are exactly the same as for $\vec{a}'$. We only need to consider the second sub-case, in which the index $\ell$ is greater than or equal to the index of $b_2$ but smaller than that of $b_3$ in $\vec{b}$. In other words, either $\ell = i_r$ is greater than the index of $b_2$ or $\ell$ is precisely the index of $b_2$ in $\vec{a}'$. If the latter occurs, let $i_r$ be the end-point of the run preceding $b_2$ in $\vec{b}$. To prove that (C1) holds for $\vec{b}$ at $\ell$, we aim to show that
\[
b_2 + B_r \triangleq b_2 + \sum_{j=4}^{i_r} b_j \geq \sum_{j=1}^{i_r-2} 2^j, \tag{17}
\]
noting that in this sub-case $\{b_2\} \cup \{b_j: 4 \leq j \leq i_r\}$ forms the set of the first $i_r - 2$ elements in $\vec{b}$. We demonstrate below that (17) can be implied from (10) in both cases when $i_r = 4$ and $i_r > 4$. First, assume that $i_r = 4$, i.e., $r = 1$ and the run $R_1$ consists of only one index 4. Now, (10) can be written as
\[c_1 + c_2 + c_3 + c_4 \geq 4 \sum_{j=1}^{2} 2^j = 30,
\]
and what we need to prove is
\[b_1 + \left\lfloor \frac{c_4}{2} \right\rfloor \geq \sum_{j=1}^{2} 2^j = 6,
\]
noting that the element in $\vec{b}$ corresponding to $c_4$ is either $\lfloor c_4/2 \rfloor$ or $\lceil c_4/2 \rceil$. Equivalently, plugging in the formula for $b_2$, what we aim to show is
\[(c_1 - 1) + \left\lfloor \frac{c_4}{2} \right\rfloor \geq 6,
\]
which is correct because \( c_1 \geq 3 \) and \( c_4 \geq 8 \), given that \( c_4 \geq c_3 \geq c_2 \geq c_1 \) and \( c_4 + c_3 + c_2 + c_1 \geq 30 \). To complete this sub-case, we assume that \( i_r > 4 \). In this scenario, \( C_r \) in (10) has at least two terms \( c_j \)'s, which are all greater than or equal to \( c_2 \) and \( c_3 \), and hence, \( C_r \geq c_2 + c_3 \). Therefore, using Lemma 7, we have

\[
b_2 + B_r = (c_1 - 1) + B_r > \frac{c_1}{4} + \left( \left\lfloor \frac{C_r}{2} \right\rfloor - 1 \right) \geq \frac{c_1}{4} + \left( \frac{C_r - 1}{2} - 1 \right) = \frac{c_1}{4} + \left( \frac{C_r}{2} - \frac{3}{2} \right)
\]

\[
= \frac{1}{4}(c_1 + 2C_r) - \frac{3}{2} \geq \frac{1}{4}(c_1 + c_2 + c_3 + C_r) - \frac{3}{2} \geq \frac{1}{4}\sum_{j=1}^{i_r} 2^j - \frac{3}{2} = \frac{i_r - 2}{2}.
\]

Thus, (17) follows.

- (Case b2) \( b_2 > b_3 \). The sorted sequence \( \vec{b}' \) has the following format.

```
runs   b_3   runs   b_2   runs
```

Note that it is possible that there are no runs before \( b_3 \), or between \( b_3 \) and \( b_2 \), or after \( b_2 \).

Due to Lemma 4, we only need to demonstrate that (C1) holds for \( \vec{b}' \) at the end-point \( i_r \) of each run \( R_r \), \( 1 \leq r \leq k \), and at \( b_2 \) and \( b_3 \). Similar to Case b1, we only need to investigate the second sub-case when the index of interest \( \ell \) is greater than or equal to the index of \( b_3 \) but smaller than that of \( b_2 \) in \( \vec{b}' \). In other words, either \( \ell = i_r \) is greater than the index of \( b_2 \) and smaller than that of \( b_2 \), or \( \ell \) is precisely the index of \( b_3 \) in \( \vec{b}' \). If the latter occurs, let \( i_r \) be the end-point of the run preceding \( b_3 \) in \( \vec{b}' \). To prove that (C1) holds for \( \vec{b}' \) at \( \ell \), we aim to show that

\[
b_3 + B_r \triangleq b_3 + \sum_{j=4}^{i_r} b_j \geq \sum_{j=1}^{i_r-2} 2^j,
\]

(18)

noting that in this sub-case \( \{b_3\} \cup \{b_j : 4 \leq j \leq i_r\} \) forms the set of the first \( i_r - 2 \) elements in \( \vec{b}' \). We demonstrate below that (18) can be implied from (10) in both cases when \( i_r = 4 \) and \( i_r > 4 \). First, assume that \( i_r = 4 \), i.e., \( r = 1 \) and the run \( R_1 \) consists of only one index 4. Now, (10) can be written as

\[
c_1 + c_2 + c_3 + c_4 \geq 4 \sum_{j=1}^{2} 2^j = 30,
\]

and what we need to prove is

\[
b_3 + \left\lfloor \frac{c_4}{2} \right\rfloor \geq \sum_{j=1}^{2} 2^j = 6,
\]

noting that the element in \( \vec{b}' \) corresponding to \( c_4 \) is either \( \lfloor c_4/2 \rfloor \) or \( \lceil c_4/2 \rceil \). Equivalently, plugging in the formula for \( b_3 \), what we aim to show is

\[
c_2 - c_1 + \left\lfloor \frac{c_3 + c_1 - c_2}{2} \right\rfloor + \left\lfloor \frac{c_4}{2} \right\rfloor \geq 6.
\]

This inequality is correct because

\[
c_2 - c_1 + \left\lfloor \frac{c_3 + c_1 - c_2}{2} \right\rfloor + \left\lfloor \frac{c_4}{2} \right\rfloor \geq c_2 - c_1 + \frac{c_3 + c_1 - c_2 - 1}{2} + \frac{c_4 - 1}{2} \geq \frac{c_3 + c_4}{2} - 1 > 6,
\]

where the last inequality holds because \( c_3 + c_4 \geq 16 \), given that \( c_4 \geq c_3 \geq c_2 \geq c_1 \) and \( c_1 + c_2 + c_3 + c_4 \geq 30 \). To complete this sub-case, we assume that \( i_r > 4 \). In this scenario, \( C_r \)
in (10) has at least two terms $c_j$’s, which are all greater than or equal to $c_1$ and $c_2$, and hence, $C_r \geq c_1 + c_2$. Therefore, using Lemma [7] and (10), we have

$$b_3 + B_r = \left( c_2 - c_1 + \left\lfloor \frac{c_3 + c_1 - c_2}{2} \right\rfloor \right) + B_r \geq \frac{c_2 + c_3 - c_1 - 1}{2} + \left( \left\lfloor \frac{C_r}{2} \right\rfloor - 1 \right)$$

$$\geq \frac{c_3 - 1}{2} + \left( \frac{C_r - 1}{2} - 1 \right) > \frac{c_3}{4} + \frac{C_r}{2} - \frac{3}{2}$$

$$\geq \frac{1}{4}(c_3 + 2C_r) - \frac{3}{2} \geq \frac{1}{4}(c_1 + c_2 + c_3 + C_r) - \frac{3}{2} \geq \frac{1}{4}\sum_{j=1}^{i_r} 2^j - \frac{3}{2} = \sum_{j=1}^{i_r-2} 2^j.$$ 

Hence, (18) follows. Thus, we have shown that $\vec{b}'$ also satisfies (C1) and is $(h-1)$-feasible. \qed