Optimal Prefix Codes with Fewer Distinct Codeword Lengths are Easier to Construct*

Ahmed Belal † Amr Elmasry ‡

Abstract

A new method for constructing minimum-redundancy binary prefix codes is described. Our method does not explicitly build a Huffman tree; instead it uses a property of optimal prefix codes to compute the codeword lengths corresponding to the input weights. Let n be the number of weights and k be the number of distinct codeword lengths. The running time of our algorithm is $O(16^k \cdot n)$, which is asymptotically faster than Huffman's algorithm for sufficiently small k. If the given weights were presorted, our algorithm requires $O(9^k \cdot \log^{2k-1} n)$ comparisons, which is sub-linear for sufficiently small k.

1 Introduction

Minimum-redundancy coding plays an important role in data compression applications [15], as it gives the best possible compression of a finite text when using one static code per alphabet symbol. Hence, this encoding method is extensively used in various fields of computer science like picture compression, data transmission, etc. In accordance, the methods used for calculating minimum-redundancy prefix codes that correspond to sets of input symbol weights are of great interest [3, 10, 8, 11].

The minimum-redundancy prefix code problem is to determine, for a given list $W = [w_1, \ldots, w_n]$ of n positive symbol weights, a list $L = [l_1, \ldots, l_n]$ of n corresponding integer codeword lengths such that $\sum_{i=1}^{n} 2^{-l_i} \leq 1$, and $\sum_{i=1}^{n} w_i l_i$ is minimized. (Throughout the paper, when we mention that Kraft inequality is satisfied, it is satisfied with an equality, i.e., $\sum_{i=1}^{n} 2^{-l_i} = 1$.) Once we have the codeword lengths corresponding to a given list of symbol weights, constructing a corresponding prefix code can be easily done in linear time using standard techniques.

Finding a minimum-redundancy code for $W = [w_1, \ldots, w_n]$ is equivalent to finding a binary tree with minimum-weight external path length $\sum_{i=1}^{n} w(x_i)l(x_i)$ among

*A preliminary version of the paper appeared in the 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS) 2006 [2].

†Department of Computer and systems Engineering, Alexandria University, Egypt

‡Datalogisk Institut, University of Copenhagen, Denmark

all binary trees with leaves $x_1, \ldots, x_n$, where $w(x_i) = w_i$, and $l(x_i) = l_i$ is the depth of x_i in the corresponding tree. Hence, if we consider a leaf as a weighted node, the minimum-redundancy prefix code problem can be defined as the problem of constructing such an optimal binary tree for a given set of weighted leaves.

Based on a greedy approach, Huffman's algorithm [7] constructs specific optimal trees, which are referred to as Huffman trees. While every Huffman code is an optimal prefix code, the converse is not true. The Huffman algorithm starts with a list $\mathcal{H}$ of n nodes whose values correspond to the given n weights. In the general step, the algorithm selects the two nodes with the smallest values in the current list and removes them from $\mathcal{H}$. Next, the removed nodes become children of a new internal node, which is inserted in $\mathcal{H}$. This internal node is assigned a value equals the sum of the values of its children. The general step is repeated until there is only one node in $\mathcal{H}$, the root of the Huffman tree. The internal nodes of a Huffman tree are thereby assigned values throughout the algorithm; the value of an internal node is the sum of the weights of the leaves of its subtree. The Huffman algorithm requires $O(n \log n)$ time and linear space. The Huffman's algorithm can be implemented in linear time if the input list was presorted [13].

A distribution-sensitive algorithm is an algorithm whose performance relies on how the distribution of the input affects the output [9, 12]. For example, a related such algorithm is that of Moffat and Turpin [11]; where they show how to construct an optimal prefix code on a sorted-by-weight alphabet of n symbols, which includes r distinct symbol weights, in $O(r+r \log(n/r))$ time. Alternatively, an output-sensitive algorithm is an algorithm whose performance relies on properties of its output [4]. The algorithms proposed in [8] are in a sense output sensitive, since their additional space complexities depend on the maximum codeword length l of the output code; the B-LazyHuff algorithm [8] runs in $O(n)$ time and requires $O(l)$ extra storage to construct an optimal prefix code on a sorted-by-weight n-symbol alphabet.

In this paper, we give an output-sensitive algorithm for constructing minimum-redundancy prefix codes; our algorithm's performance depends on, k, the number of different codeword lengths (the number of levels that have leaves in the corresponding optimal binary tree). We distinguish two cases: the so called *sorted case*, if the sequence of input weights is presorted, and the *unsorted case*, otherwise. For the unsorted case, the running time of our algorithm is $O(16^k \cdot n)$, which is linear when k is a constant. For the sorted case, our algorithm uses $O(9^k \cdot \log^{2k-1} n)$ comparisons, which is sub-linear for sufficiently small k.

Throughout the paper, we interchangeably use the terms leaves and weights. The number of distinct codeword lengths of a prefix code is the same as the number of levels that contain leaves in the corresponding weighted tree. Unless otherwise stated, we assume that the input weights are unsorted. Unless explicitly mentioned, a node of a tree can be either a leaf or an internal node. The levels of the tree are numbered bottom up starting from level 0, i.e., the root has the highest level number l, its children are at level $l-1$, and the leaves furthest from the root are at

level 0 (note that this level numbering is nonstandard); the length of the codeword assigned to a weight at level j is accordingly equal to $l - j$.

The paper is organized as follows. In Section 2, we give a property of optimal prefix-code trees on which our construction algorithm relies. In Section 3, we give the basic algorithm and prove its correctness. We show in Section 4 how to implement the basic algorithm of Section 3 to ensure the output-sensitive behavior; consequently, the claimed bounds are driven. We conclude the paper in Section 5.

2 The exclusion property

Consider a a list of n weights and a binary tree T that has the following properties:

1. The n leaves of T correspond to the given n weights.
2. The value of an internal node of T equals the sum of the weights of the leaves of its subtree.
3. For any level of T, let $y_1, y_2, y_3 \ldots$ be the nodes of that level in non-decreasing order with respect to their values. Then, y_{2i-1} and y_{2i} are siblings for all $i \geq 1$.

We define the *exclusion property* [1, 2] for T as follows: T has the exclusion property if and only if the values of the nodes at a level are not smaller than the values of the nodes at the lower levels.

Lemma 1 *[1] Given a prefix code whose corresponding tree T has the aforementioned properties, the given prefix code is optimal and T is a Huffman tree if and only if T has the exclusion property.*

Proof. First, assume that T does not have the exclusion property. It follows that there exist two nodes y and y' respectively at levels η and η' such that $\eta < \eta'$ and $value(y) > value(y')$. Swapping the subtree of y with the subtree of y' results in another tree with a smaller external path length and a different list of levels, implying that the given prefix code is not optimal.

Next, assume that T has the exclusion property. Let $[x_1, \ldots, x_n]$ be the list of leaves of T, with $w(x_i) \leq w(x_{i+1})$ for all $i \geq 1$. We prove by induction on the number of leaves n that T is an optimal binary tree, which corresponds to an optimal prefix code. The base case follows trivially when $n = 2$. As a result of the exclusion property, the two leaves x_1, x_2 must be at the lowest level of T. Also, property 3 of T implies that these two leaves are siblings. Alternatively, there is an optimal binary tree with leaves $[x_1, \ldots, x_n]$ where x_1 and x_2 are siblings; a fact that is used to prove the correctness of Huffman's algorithm [7]. Remove x_1, x_2 from T, replace their parent with a leaf x_{12} whose weight equals $w(x_1) + w(x_2)$, and let T' be the resulting tree. Since T' has the exclusion property, it follows using induction

that T' is an optimal tree with respect to its $n-1$ leaves $[x_{12}, x_3, \ldots, x_n]$. Hence, T corresponds to an optimal prefix code. Our property 3 ensures that every two consecutive nodes in the non-decreasing order of values are siblings, which is the way Huffman's algorithm works. It follows that T is a Huffman tree. □

Assume that a set of nodes of a tree that corresponds to a prefix code are numbered, starting from 1, in a non-decreasing order by their values. We define the *rank* of a node to be its number according to this ordering. The *sibling property* was introduced by Gallager [6, 14]. The sibling property states that a tree that corresponds to a prefix code is a Huffman tree if and only if the nodes with ranks $2i-1$ and $2i$ are siblings for all $i \geq 1$. In fact, the sibling property is equivalent to property 3 combined with the exclusion property. This equivalence can be directly proven, indicating that a tree T that has the exclusion property is a Huffman tree.

In general, building a tree T that has the exclusion property (Huffman tree) by evaluating all its internal nodes requires $\Omega(n \log n)$. This follows from the fact that knowing the values of the internal nodes of T implies knowing the sorted order of the input weights, a problem that requires $\Omega(n \log n)$ in the comparison-based decision-tree model. Our main idea is that we do not have to explicitly construct T in order to find optimal codeword lengths. Instead, we only need to find the values for some of —and not all— the internal nodes to maintain the exclusion property.

3 The basic construction method

Given a list of weights, we build a corresponding optimal tree level by level in a bottom-up manner. Starting with the lowest level (level 0), a weight is momentarily assigned to a level as long as its value is less than the sum of the two nodes with the currently smallest ranks at that level; this ensures the exclusion property. Kraft inequality is enforced by making sure that, at the end of the algorithm, the number of nodes at every level is even, and that the number of nodes at the highest level containing leaves is a power of two. This results in some weights being moved upwards from their initially-assigned levels to the higher levels. The details follow.

3.1 An illustrative example

For the sake of illustration, we start with an example. Consider a list with thirty weights: ten weights of value 2, ten of value 3, five of value 5, and five of value 9.

To construct the optimal codes, we start by finding the smallest two weights in the list; these have the values $2, 2$. We now identify all the weights in the list with value less than 4, the sum of these two smallest weights. There are twenty such weights: ten weights of value 2 and ten of value 3. All these weights are momentarily assigned to the bottom level, that is level 0 with respect to our level numbering. The number of nodes at level 0 is now even; so, we go to the next upper level (level

1). We identify the smallest two nodes at level 1, amongst the two smallest internal nodes resulting from combining nodes already at level 0 (these have values $4, 4$) and the two smallest weights among those remaining in the list (these have values $5, 5$). It follows that the smallest two nodes at level 1 will be the two internal nodes $4, 4$ whose sum is 8. All the remaining weights with value less than 8 are to be momentarily assigned to level 1. Accordingly, level 1 now contains an odd number of nodes: ten internal nodes and five weights of value 5. See Figure 1(a).

To make the number of nodes at level 1 even, we move the node with the largest rank at level 1 to the, still empty, next upper level (level 2). The node to be moved, in this case, is an internal node with value 6. Moving an internal node one level up implies moving the weights in its subtree one level up. So, the subtree consisting of the two weights of value 3 is moved one level up. At the end of this stage, level 0 contains ten weights of value 2 and eight weights of value 3; level 1 contains two weights of value 3 and five weights of value 5. See Figure 1(b).

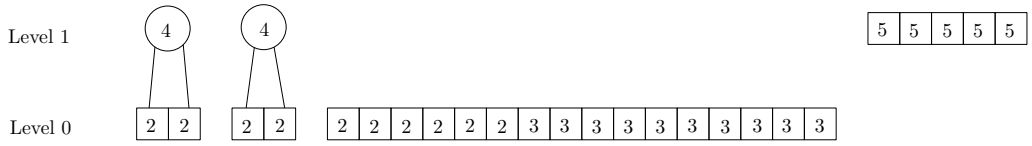
The currently smallest two internal nodes at level 2 have values $6, 8$ and the smallest weight in the list has value 9. This means that all the five remaining weights in the list will be momentarily assigned to level 2. Now, level 2 contains eight internal nodes and five weights, for a total of thirteen nodes. See Figure 1(c).

Since we are done with all the weights, we only need to enforce the condition that the number of nodes at level 3 is a power of two. All we need to do is to move the three nodes with the largest ranks, from level 2, one level up. The largest three nodes at level 2 are the three internal nodes of values $12, 12$ and 10. So, we move eight weights of value 3 and two weights of value 5 one level up. As a result, the number of nodes at level 3 will be 8; that is a power of two. The root will then be at level 6. The final distribution of weights will be: ten weights of value 2 at level 0; ten weights of value 3 and three weights of value 5 at level 1; and the remaining weights, two of value 5 and five of value 9, at level 2. The corresponding codeword lengths are 6, 5 and 4 respectively. See Figure 1(d).

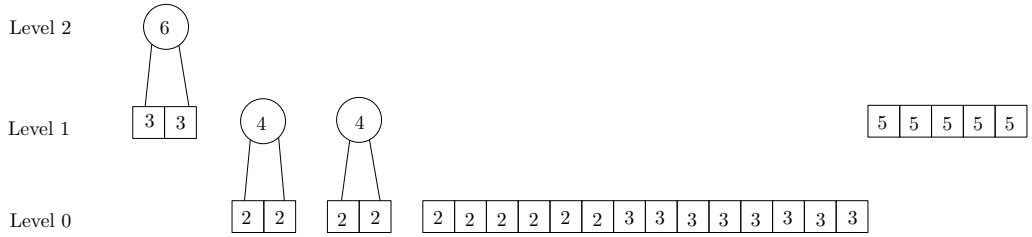
Note that we have not included all the internal nodes within Figure 1. We have consciously only drawn the internal nodes that are required to be evaluated by our algorithm; this will be elaborated throughout the next subsections.

3.2 The algorithm

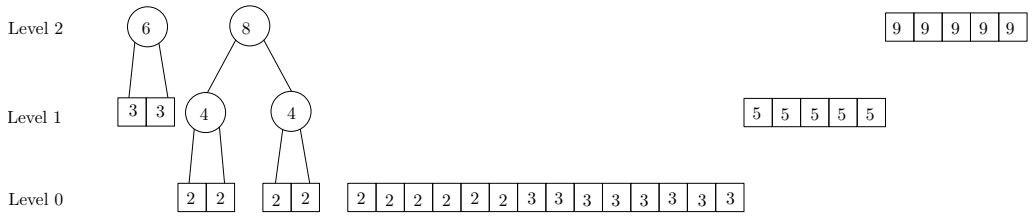
The idea of the algorithm should be clear. We construct an optimal tree by maintaining the exclusion property for all the levels. Once the weights are placed in such a way that the exclusion property is satisfied, the property will as well be satisfied among the internal nodes. Adjusting the number of nodes at each level will not affect the exclusion property, since we are always moving the largest nodes from one level to the next higher level. A formal description follows. (Note that the main ideas of our basic algorithm described in this subsection are pretty similar to those of the Lazy-Traversal algorithm described in [8].)



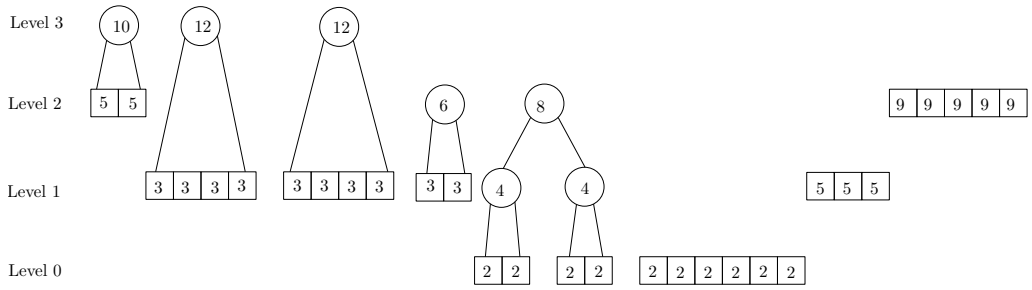
(a) Initial assignment for levels 0 and 1



(b) Moving the node with the largest rank at level 1 to level 2



(c) Initial assignment for level 2



(d) The final weight assignments

Figure 1: The basic construction method. Example 3.1. Rectangles represent leaves and circles represent internal nodes.

1. Consider a list of input symbol weights W (not necessarily sorted). The smallest two weights are found, removed from W, and placed at the lowest level 0; their sum S is computed. The list W is scanned and all weights less than S are removed from W and placed at level 0. Set $\eta \leftarrow 0$.

2. Repeat the following steps until W is empty:

   (a) If the number of nodes at level η is odd, move the subtree rooted at the node with the largest rank from level η to level $\eta + 1$.

   (b) Determine the new weights that will go to level $\eta+1$ as follows. Find the smallest two internal nodes at level $\eta + 1$, and the smallest two weights among those remaining in W. Find the smallest two values amongst these four, and let their sum be S. Scan W for all weights less than S, and move them to level $\eta + 1$.

   (c) $\eta \longleftarrow \eta + 1$.

3. Set $\hat{\eta} \leftarrow \eta$, i.e., $\hat{\eta}$ is the highest level that is currently assigned weights. Let m be the current number of nodes at level $\hat{\eta}$. Move the $2^{\lceil \lg m \rceil} - m$ subtrees rooted at the nodes with the largest ranks from level $\hat{\eta}$ to level $\hat{\eta} + 1$.

3.3 Proof of correctness

To guarantee its optimality following Lemma 1, we need to show that both Kraft inequality and the exclusion property hold for the constructed tree.

Maintaining Kraft inequality. First, we show by induction that the number of nodes at every level of the tree is even. Assume that this is true up to level $\eta - 1$. Since any subtree of our tree is a full binary tree (every node has zero or two children), the number of nodes per level within any subtree is even except for its root. At step $2(a)$ of the algorithm, if the number of nodes at level η is odd, we move a subtree one level up. We are thus moving even numbers of nodes between levels among the lower $\eta - 1$ levels. Hence, the number of nodes remains even per level among those levels. On the other hand, the number of nodes at level η either decreases by 1 (if the moved root has no children) or increases by 1 (if the moved root has two children). Either way, the number of nodes at level η becomes even.

Second, we show that the number of nodes at the last level that is assigned weights is a power of two. At step 3 of the algorithm, if m is a power of two, no subtrees are moved up and Kraft inequality holds. Otherwise, we move $2^{\lceil \lg m \rceil} - m$ nodes from level $\hat{\eta}$ to level $\hat{\eta}+1$, leaving $2m - 2^{\lceil \lg m \rceil}$ nodes at level $\hat{\eta}$ other than the children of the roots that have just been moved to level $\hat{\eta} + 1$. Now, the number of nodes at level $\hat{\eta} + 1$ is $m - 2^{\lceil \lg m \rceil - 1}$ internal nodes resulting from combining pairs of nodes from level $\hat{\eta}$, plus the $2^{\lceil \lg m \rceil} - m$ nodes that we have just moved. This sums up to $2^{\lceil \lg m \rceil - 1}$ nodes; that is a power of two.

Maintaining the exclusion property. We prove by induction that the —even stronger— sibling property holds, as long as we are correctly evaluating the prescribed internal nodes following Huffman's rules. Assume that the sibling property holds up to level $\eta - 1$. Throughout the algorithm, we maintain the property by making sure that the sum of the values of the two nodes with the smallest ranks at a level is larger than all the values of the nodes at this level. When we move a subtree from level $\eta - 1$ one level up, the root of this subtree is the node with the largest rank at its level. The validity of the sibling property at the lowest $\eta - 1$ levels implies that the children of the node with the largest rank at a level have the largest ranks among the nodes at their level. Hence, all the nodes of the moved subtrees at a certain level must have had the largest ranks among the nodes of their level. When such nodes are moved one level up their values are thus larger than those of the nodes at the lower levels, and the exclusion property is still maintained.

3.4 Discussion

We are still far from being done yet. Though we have introduced the main idea behind the algorithm, some crucial details are still missing. Namely, we did not show how to evaluate the essential internal nodes. Should we have to evaluate all the internal nodes, breaking the $n \log n$ bound would have been impossible. Fortunately, we only need to evaluate few internal nodes per level. More precisely, except for the last level that is assigned weights, we may need to evaluate three internal nodes per level: the two with the smallest ranks and the one with the largest rank. The challenge is how to do that efficiently. Another pitfall of our basic method, as we have just described, is that we are to evaluate internal nodes for every level of the tree up to the last level that is assigned weights. The work would thus be proportional to the difference between the length of the maximum and the minimum codeword lengths. We still need to do better, and the way out is to skip doing work at the levels that will not be assigned any weights. Again, the challenge is how to do that efficiently. Subsequently, we thus explain our detailed construction in Section 4.

4 The detailed construction method

Up to this point, we have not shown how to evaluate the internal nodes needed by our basic algorithm, and how to search within the list W to decide which weights are to be assigned to which levels. The basic intuition behind the novelty of our approach is that it does not require evaluating all the internal nodes of the tree corresponding to the prefix code, and would thus surpass the $n \log n$ bound for several cases. In this section, we show how to implement the basic algorithm in an output-sensitive behavior, filling in the missing details.

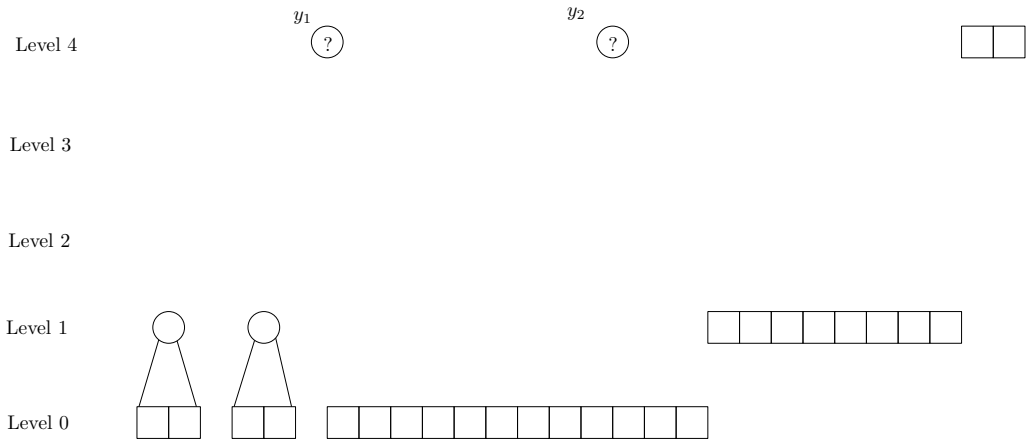


Figure 2: Example 4.1. with $n = 16$.

4.1 An illustrative example

The main idea is clarified through an example with $3n/2 + 2$ weights, where n is a power of two. Assume that the resulting optimal tree will turn out to have $k = 3$: n leaves at level 0, $n/2$ at level 1, and two at level $\lg n$. Note that the $3n/2$ leaves at levels 0 and 1 combine to produce two internal nodes at level $\lg n$. It is straightforward to come up with a set of weights that fulfills this outcome. However, to illustrate how the algorithm works for any such set of weights, it is better to handle the situation without explicitly deciding the weights.

For such case, we show how to apply our algorithm so that the optimal codeword lengths are produced in linear time, even if the weights were not presorted. Determining the weights to be assigned to level 0 can be easily done by finding the smallest two weights and scanning through the list of weights. To determine the weights to be assigned to level 1, we need to find the values of the smallest two internal nodes at level 1; these are respectively the sum of the smallest two pairs of weights. At this point, lets assume that the algorithm uses an oracle that recommends checking level $\lg n$ next.

A more involved task is to evaluate the two internal nodes y_1 and y_2 at level $\lg n$, which amounts to identifying the smallest as well as the largest $n/2$ nodes amongst the n nodes at level 1. The main advantage is that we do not need to sort the values of the nodes at level 1. In addition, we do not need to explicitly evaluate all the $n/2$ internal nodes at level 1 resulting from the pairwise combinations of the n weights at level 0. What we really need to know is the sum of the smaller half as well as the sum of the larger half among the nodes at level 1. See Figure 2. We show next that evaluating y_1 and y_2 can be done in linear time via a simple pruning procedure.

The nodes at level 1 consist of two sets; one set has $n/2$ leaves whose weights are known and thus their median M can be found in linear time [5], and another set containing $n/2$ internal nodes whose values are not known, but whose median

M' can still be computed in linear time by simply finding the two middle weights of the n leaves at level 0 and adding them. Assuming without loss of generality that $M > M'$, then the larger half of the $n/2$ weights at level 1 contribute to y_2, and the smaller half of the n weights at level 0 contribute to y_1. The above step of finding new medians for the leftovers of the two sets is repeated recursively on a problem half the size. This results in a procedure with a running time that satisfies the recurrence $T(n) = T(n/2) + O(n)$, whose solution results in $T(n) = O(n)$.

If the list of weights was presorted, no comparisons are required to find M or M'; we only need to compare them together. The total number of comparisons needed satisfies the recurrence $C_s(n) = C_s(n/2) + O(1)$, and hence $C_s(n) = O(\log n)$.

4.2 Overview

Following the basic construction method, the optimal tree will be built bottom up. However, this will not be done level by level; we shall only work in the vicinity of the levels that will end up having leaves. Once we finish assigning weights to a level, we should be able to decide about the number of the next higher level to consider (Section 4.3.3). As stated earlier, throughout the algorithm we have to efficiently evaluate some internal nodes. In general, for a specified level and a fixed t, we need to be able to evaluate the node with the t-th smallest or largest rank among the internal nodes, or even among all the nodes, at that level (Section 4.3.2). To be able to do that efficiently, we shall illustrate a method to evaluate one specific internal node that serves as the median of the nodes on the specified level. More precisely, if one counts the number of weights contributing to each node at the specified level, and supposedly sort the list of nodes on that level by value, the sought node will be the node that splits this sorted list in two sublists the number of weights contributing to each is close as possible to the other. We shall show how such splitting procedure is accomplished in a recursive manner (Section 4.3.1). Once we are able to evaluate such node, the other tasks —like finding the next level to be assigned weights and finding the t-th smallest or largest nodes— can be done by repeatedly calling the splitting procedure. The details are to come up next.

4.3 The algorithm

Let $\eta_1 = 0 < \eta_2 < \ldots \eta_j$ be the levels that have already been assigned weights after some iterations of our algorithm (other levels only have internal nodes). Let n_i be the number of leaves so far assigned to level η_i, and $N_j = \sum_{i=1}^{j} n_i$.

At the current iteration, we are looking forward to compute, η_{j+1}, the level that will next be assigned weights by our algorithm. We use the fact that the weights that have already been assigned up to level η_j are the only weights that may contribute to the values of the internal nodes below and up to level η_{j+1}.

Consider the internal node χ'_j at level η_j, where the sum of the number of leaves

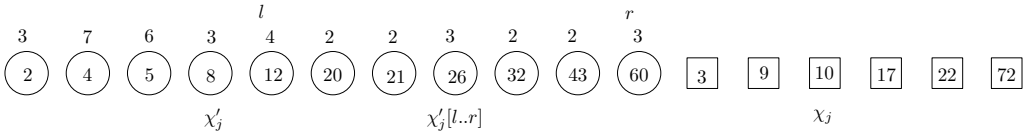


Figure 3: Nodes χ'_j, $\chi'_j[l..r]$ and χ_j at level η_j. Numbers inside nodes represent their values, and those outside internal nodes represent their multiplicity. For illustration purposes, the internal nodes are drawn sorted by value and so are the leaves.

in the subtrees of level-η_j internal nodes whose ranks are smaller than that of χ'_j is at most but closest to $N_{j-1}/2$. We call χ'_j the *splitting node of the internal nodes* at level η_j. In other words, if we define the *multiplicity* of a node to be the number of leaves in its subtree, then χ'_j is the weighted-by-multiplicity median within the sorted-by-value sequence of the internal nodes at level η_j.

Analogously, consider the node χ_j (not necessarily an internal node) at level η_j, where the sum of the number of leaves in the subtrees of level-η_j internal nodes whose ranks are smaller than that of χ_j plus the number of level-η_j leaves whose ranks are smaller than that of χ_j is at most but closest to $N_j/2$. We call χ_j the *splitting node of all nodes* at level η_j. Informally, χ'_j splits the weights below level η_j in two groups having almost equal counts, and χ_j splits the weights below and up to level η_j in two groups having almost equal counts.

We extend the notion of splitting nodes to subsets of nodes. Let L be a list of weights constituting the leaves of a subset of the internal nodes at level η_j having consecutive ranks. The splitting node $\chi'_j(L)$ is defined as the weighted-by-multiplicity median within the sorted-by-value sequence of those internal nodes. Let L' be a subset of leaves at level η_j having consecutive ranks. The splitting node $\chi_j(L, L')$ is defined as the weighted-by-multiplicity median within the sorted-by-value sequence of L' in addition to the internal nodes at level η_j whose subtrees have the leaves L.

Figure 3 illustrates those definitions: The internal node with value 8 is the splitting node χ'_j of the internal nodes; the sum of the multiplicity of the internal nodes with values smaller than 8 is 16, and the sum of the multiplicity of those with values larger than 8 is 18. The leaf with value 10 is the splitting node χ_j of all nodes; the sum of the multiplicity of the nodes with values smaller than 10 is 21 (19 contribute to internal nodes plus 2 leaves), and the sum of the multiplicity of those with values larger than 10 is 21 (18 contribute to internal nodes plus 3 leaves).

4.3.1 Finding the splitting node

To find the splitting node χ_j of all nodes at level η_j, we repeatedly identify splitting nodes of subsets of the internal nodes. The main idea is to apply a pruning procedure in a manner similar to binary search. Starting with all the leaves and internal nodes at level η_j, we repeatedly perform the following steps: compare the median of the leaves and the splitting node of the internal nodes, discard sublists of nodes

from further consideration, and compute the median of the remaining leaves or the splitting node of the remaining internal nodes. The details follow.

We find the leaf M with the median weight among the list L of the n_j weights assigned to level η_j, and partition L into three sublists $\{M\}$, L_1 and L_2 around M, where L_1 is the list of weights with the smaller ranks. We recursively find the splitting node χ'_j of the internal nodes at level η_j using the list L' of the N_{j-1} weights at the levels below η_j, and partition L' into three sublists $L_{\chi'_j}$, L'_1 and L'_2 around χ'_j, where L'_1 is the list of weights contributing to the internal nodes with the smaller ranks. We use the function *add-weights* to compute the value of χ'_j by adding the list of weights $L_{\chi'_j}$ constituting the leaves of the subtree of χ'_j. Comparing the values of M and χ'_j, assume without loss of generality that $M > \chi'_j$. We conclude that either the weights in L_2 in addition to M must have larger ranks than χ_j, or the internal nodes corresponding to the weights in L'_1 in addition to χ'_j must have smaller ranks than χ_j. Accordingly, we either set L to L_1, and find the new median M and the new lists L_1 and L_2, or set L' to L'_2, and find the new splitting node χ'_j and the new lists L'_1 and L'_2. The pruning procedure continues until only the weights contributing to χ_j remain. As a byproduct, we compute, O_1 and O_2, the two lists of weights contributing to the nodes at level η_j whose ranks are smaller and respectively larger than that of χ_j. We also compute, *pos*, the rank of χ_j among the nodes at level η_j. See the pseudo-code of Algorithm 1 for the detailed description.

Next, consider the problem of finding the splitting node χ'_j of the internal nodes at level η_j. Observe that χ_{j-1} is a descendant of χ'_j; so, we start by recursively finding the node χ_{j-1}. Let α be the rank of χ_{j-1} among the nodes at level η_{j-1}; the numbering starts from 1. Knowing that exactly $\lambda = 2^{\eta_j - \eta_{j-1}}$ nodes from level η_{j-1} contribute to every internal node at level η_j, we conclude that the largest $\beta = (\alpha - 1) - \lambda \cdot \lfloor (\alpha - 1)/\lambda \rfloor$ among the $\alpha - 1$ nodes whose ranks are smaller than χ_{j-1}, and the smallest $\lambda - \beta - 1$ nodes among those whose ranks are larger than χ_{j-1}, are the nodes contributing to χ'_j. We proceed by finding such nodes, a procedure that requires recursively finding more splitting nodes at level η_{j-1}, in a way that will be illustrated in the next subsection. To summarize, the splitting node χ'_j of level η_j is evaluated as follows. The aforementioned pruning procedure is applied to split the weights already assigned to the lower $j-1$ levels to three groups; those contributing to χ_{j-1}, those contributing to the nodes smaller than χ_{j-1} at level η_{j-1}, and those contributing to the nodes larger than χ_{j-1} at level η_{j-1}. The weights contributing to χ'_j are: the weights of the first group, the weights among the second group contributing to the largest β nodes smaller than χ_{j-1}, and the weights among the third group contributing to the smallest $\lambda - \beta - 1$ nodes larger than χ_{j-1}. We also compute, *pos*, the rank of χ'_j among the internal nodes at level η_j. See the pseudo-code of Algorithm 2 for the detailed description.

Algorithm 1 *find-splitting-all*$(j, \mathcal{L})$

$(L, L') \leftarrow cut\ (j, \mathcal{L})$
$(M, L_1, L_2) \leftarrow find\text{-}median(L)$
if $(j = 1)$ **then**
  return $|L_1| + 1, \{M\}, L_1, L_2$
end if
$(p, L_{\chi'_j}, L'_1, L'_2) \leftarrow find\text{-}splitting\text{-}internal(j, L')$
$O_1 \leftarrow O_2 \leftarrow nil,\ \ pos \leftarrow 1,\ \ s_1 \leftarrow \lfloor N_j/2 \rfloor,\ \ s_2 \leftarrow N_j - s_1 - 1$
while ($|L| \neq 0$ and $|L'| \neq 0$) **do**
  if $(M > add\text{-}weights(L_{\chi'_j}))$ **then**
    if $(|L_1| + |L'| - |L'_2| > s_1)$ **then**
      $O_2 \leftarrow catenate(\{M\}, L_2, O_2),\ \ s_2 \leftarrow s_2 - 1 - |L_2|,\ \ L \leftarrow L_1$
      $(M, L_1, L_2) \leftarrow find\text{-}median(L)$
    else
      $O_1 \leftarrow catenate(O_1, L'_1, L_{\chi'_j}),\ \ s_1 \leftarrow s_1 - |L'_1| - |L_{\chi'_j}|,\ \ pos \leftarrow pos + p,\ \ L' \leftarrow L'_2$
      $(p, L_{\chi'_j}, L'_1, L'_2) \leftarrow find\text{-}splitting\text{-}internal(j, L')$
    end if
  else
    if $(|L_2| + |L'| - |L'_1| > s_2)$ **then**
      $O_1 \leftarrow catenate(O_1, L_1, \{M\}),\ \ s_1 \leftarrow s_1 - |L_1| - 1,\ \ pos \leftarrow pos + |L_1| + 1,\ L \leftarrow L_2$
      $(M, L_1, L_2) \leftarrow find\text{-}median(L)$
    else
      $O_2 \leftarrow catenate(L_{\chi'_j}, L'_2, O_2),\ \ s_2 \leftarrow s_2 - |L_{\chi'_j}| - |L'_2|,\ \ L' \leftarrow L'_1$
      $(p, L_{\chi'_j}, L'_1, L'_2) \leftarrow find\text{-}splitting\text{-}internal(j, L')$
    end if
  end if
end while
if $(|L| = 0)$ **then**
  while ($|L'_1| > s_1$ or $|L'_2| > s_2$) **do**
    if $(|L'_2| > s_2)$ **then**
      $O_1 \leftarrow catenate(O_1, L'_1, L_{\chi'_j}),\ \ s_1 \leftarrow s_1 - |L'_1| - |L_{\chi'_j}|,\ \ pos \leftarrow pos + p,\ \ L' \leftarrow L'_2$
    else
      $O_2 \leftarrow catenate(L_{\chi'_j}, L'_2, O_2),\ \ s_2 \leftarrow s_2 - |L_{\chi'_j}| - |L'_2|,\ \ L' \leftarrow L'_1$
    end if
    $(p, L_{\chi'_j}, L'_1, L'_2) \leftarrow find\text{-}splitting\text{-}internal(j, L')$
  end while
  $O_1 \leftarrow catenate(O_1, L'_1),\ \ O_2 \leftarrow catenate(L'_2, O_2),\ \ pos \leftarrow pos + |L'_1|$
  return $pos, L_{\chi'_j}, O_1, O_2$
else
  while ($|L_1| > s_1$ or $|L_2| > s_2$) **do**
    if $(|L_2| > s_2)$ **then**
      $O_1 \leftarrow catenate(O_1, L_1, \{M\}),\ \ s_1 \leftarrow s_1 - |L_1| - 1,\ \ pos \leftarrow pos + |L_1| + 1,\ L \leftarrow L_2$
    else
      $O_2 \leftarrow catenate(\{M\}, L_2, O_2),\ \ s_2 \leftarrow s_2 - 1 - |L_2|,\ \ L \leftarrow L_1$
    end if
    $(M, L_1, L_2) \leftarrow find\text{-}median(L)$
  end while
  $O_1 \leftarrow catenate(O_1, L_1),\ \ O_2 \leftarrow catenate(L_2, O_2),\ \ pos \leftarrow pos + |L_1|$
  return $pos, \{M\}, O_1, O_2$
end if

Algorithm 2 *find-splitting-internal*$(j, \mathcal{L})$

1: $(\alpha, L_{\chi_{j-1}}, O_1, O_2) \leftarrow$ *find-splitting-all*$(j-1, \mathcal{L})$
2: $L_1 \leftarrow L_2 \leftarrow nil$, $\lambda \leftarrow 2^{\eta_j - \eta_{j-1}}$
3: $\beta \leftarrow \alpha - 1 - \lambda \cdot \lfloor (\alpha - 1)/\lambda \rfloor$
4: **if** $(\beta \neq 0)$ **then**
5: $\quad (O_1, L_1) \leftarrow$ *find-t-largest*$(\beta, j-1, O_1)$
6: **end if**
7: **if** $(\lambda - \beta - 1 \neq 0)$ **then**
8: $\quad (L_2, O_2) \leftarrow$ *find-t-smallest*$(\lambda - \beta - 1, j-1, O_2)$
9: **end if**
10: $pos \leftarrow \lceil \alpha/\lambda \rceil$, $L_{\chi'_j} \leftarrow$ *catenate*$(L_1, L_{\chi_{j-1}}, L_2)$
11: return $pos, L_{\chi'_j}, O_1, O_2$

4.3.2 Finding the t-th smallest or largest node

Consider the node $\Im_j$ that has the t-th smallest or largest rank among the nodes at level η_j. The following recursive procedure is used to evaluate $\Im_j$.

As for the case of finding the splitting node, we find the leaf with the median weight M among the list of the n_j weights already assigned to level η_j, and evaluate the splitting node χ'_j of the internal nodes at level η_j (applying the aforementioned recursive procedure) using the list of the N_{j-1} leaves of the lower levels. Comparing M to χ'_j, we discard either M or χ'_j plus one of the four sublists —the two sublists of n_j leaves and the two sublists of N_{j-1} leaves— as not contributing to $\Im_j$. Repeating this pruning procedure, we identify the weights that contribute to $\Im_j$ and hence evaluate $\Im_j$. Accordingly, we also identify the list of weights that contribute to the nodes at level η_j whose values are smaller or lager than the value of $\Im_j$.

The main ideas of this procedure are pretty similar to those of finding the splitting node, and hence we omit the details and leave them for the reader.

4.3.3 Computing η_{j+1} (the next level that will be assigned weights)

We start by finding the minimum weight w among the weights remaining in W (not assigned to any of the first j levels), by applying the function *minimum-weight*. We next use the value of w to search within the nodes at level η_j in a manner similar to binary search. The main idea is to find the maximum number of nodes with the smallest ranks at level η_j such that the sum of their values is less than w. We find the splitting node χ_j at level η_j, and evaluate the sum of χ_j plus the weights contributing to the nodes at level η_j whose ranks are less than that of χ_j. Comparing this sum with w, we decide which sublist of the N_j leaves to proceed to find its splitting node. At the end of this searching procedure, we would have identified the weights contributing to the γ smallest nodes at level η_j, such that the sum of their values is less than w and γ is maximum. (Note that γ is at least 2.) Accordingly, the level to be considered next for assigning weights is level $\eta_j + \lfloor \lg \gamma \rfloor$. See the pseudo-code of Algorithm 3 for the detailed description.

Algorithm 3 *compute-next-level*$(j, \mathcal{L}, W)$

```
1: min ← minimum-weight(W)
2: L ← 𝓛, γ ← 0
3: while (|L| ≠ 0) do
4:     (α, L_{χ_j}, O_1, O_2) ← find-splitting-all(j, L)
5:     sum ← add-weights(O_1) + add-weights(L_{χ_j})
6:     if (sum < min) then
7:         min ← min − sum,  γ ← γ + α,  L ← O_2
8:     else
9:         L ← O_1
10:    end if
11: end while
12: η_{j+1} ← η_j + ⌊lg γ⌋
13: return η_{j+1}
```

To prove the correctness of this procedure, consider any level η where $\eta_j < \eta < \eta_j + \lfloor \lg \gamma \rfloor$. The subtrees of the two internal nodes with the smallest ranks at level η have at most $2^{\eta - \eta_j + 1} \leq 2^{\lfloor \lg \gamma \rfloor} \leq \gamma$ nodes at level η_j. Hence, the sum of the values of such two nodes is less than w. For the exclusion property to hold, no weights are to be assigned to any of these levels. On the contrary, the subtrees of the two internal nodes with the smallest ranks at level $\eta_j + \lfloor \lg \gamma \rfloor$ have more than γ nodes at level η_j, and hence the sum of their values is at least w. For the exclusion property to hold, at least the weight w is to be momentarily assigned to level $\eta_j + \lfloor \lg \gamma \rfloor$.

4.3.4 Maintaining Kraft inequality

After deciding the value of η_{j+1}, we need to maintain Kraft inequality. This is accomplished by moving the subtrees of the ν nodes with the largest ranks from level η_j one level up. Let m be the number of nodes currently at level η_j and let $\lambda = 2^{\eta_{j+1} - \eta_j}$, then the number of subtrees to be moved up is $\nu = \lambda \cdot \lceil m/\lambda \rceil - m$. See the pseudo-code of Algorithm 4. Note that when $\eta_{j+1} - \eta_j = 1$ (as in the case of our basic algorithm), then $\nu = 1$ if m is odd and $\nu = 0$ otherwise.

Algorithm 4 *maintain-Kraft-inequality*$(j, \mathcal{L}, W)$

```
1: m ← count-nodes(j)
2: if (|W| ≠ 0) then
3:     λ ← 2^{η_{j+1} − η_j}
4:     ν ← λ · ⌈m/λ⌉ − m
5: else
6:     ν ← 2^{⌈lg m⌉} − m
7: end if
8: (𝓛, L) ← find-t-largest(ν, j, 𝓛)
9: for every weight w ∈ L do
10:    level(w) ← level(w) + 1
11: end for
```

To establish the correctness of this procedure, we need to show that both Kraft inequality and the exclusion property hold. For a realizable construction, the number of nodes at level η_j has to be even, and if $\eta_{j+1} - \eta_j \neq 1$, the number of nodes at level $\eta_j + 1$ has to divide $\lambda/2$. If m divides λ, no subtrees are moved to level $\eta_j + 1$ and Kraft inequality holds. If m does not divide λ, then $\lambda \cdot \lceil m/\lambda \rceil - m$ nodes are moved to level $\eta_j + 1$, leaving $2m - \lambda \cdot \lceil m/\lambda \rceil$ nodes at level η_j other than those of the subtrees that have just been moved one level up. Now, the number of nodes at level $\eta_j + 1$ is $m - \lambda \cdot \lceil m/\lambda \rceil/2$ internal nodes resulting from the nodes of level η_j, plus the $\lambda \cdot \lceil m/\lambda \rceil - m$ nodes that we have just moved. This sums up to $\lambda \cdot \lceil m/\lambda \rceil/2$ nodes, which divides $\lambda/2$, and Kraft inequality holds. The exclusion property holds following the same argument given in Section 3.3. Kraft inequality for the highest level that is assigned leaves, i.e., when $|W| = 0$, is correctly maintained also following the argument given in Section 3.3.

4.3.5 Summary of the algorithm

1. The smallest two weights are found, moved from W to the lowest level $\eta_1 = 0$, and their sum S is computed. The rest of W is searched for weights less than S, which are moved to level 0 as well. Set $j \leftarrow 1$.

2. Repeat the following steps until W is empty:

   (a) Compute η_{j+1} (the next level that will be assigned weights).

   (b) Maintain Kraft inequality at level η_j (by moving the ν subtrees with the largest ranks from this level one level up).

   (c) Find the values of the smallest two internal nodes at level η_{j+1}, and the smallest two weights from those remaining in W. Find the two nodes with the smallest ranks among these four, and let their sum be S.

   (d) Search the rest of W, and assign the weights less than S to level η_{j+1}.

   (e) $j \longleftarrow j + 1$.

3. Let m be the current number of nodes at level η_j. Move the $2^{\lceil \lg m \rceil} - m$ subtrees rooted at the nodes with the largest ranks from level η_j to level $\eta_j + 1$.

4.4 Complexity analysis

Let $T(j, |\mathcal{L}|)$ be the time required to find the splitting node (and also for the t-th smallest or largest node) of a set of nodes at level η_j that are roots of subtrees having the list of leaves $\mathcal{L}$. It follows that $T(j, N_j)$ is the time required to find χ_j. Let $T'(j, |\mathcal{L}|)$ be the time required to find the splitting node of a set of internal nodes at level η_j that are roots of subtrees having the list of leaves $\mathcal{L}$. It follows that $T'(j, N_{j-1})$ is the time required to find χ'_j.

First, consider Algorithm 1. The total amount of work, in all the recursive calls, required to find the medians among the n_j weights assigned to level η_j is $O(n_j)$. During the pruning procedure to locate χ_j, the time for the i-th recursive call to find a splitting node of internal nodes at level η_j is at most $T'(j, \lfloor N_{j-1}/2^{i-1} \rfloor)$. The pruning procedure, therefore, requires at most $\sum_{i=1}^{\hat{i}} T'(j, \lfloor N_{j-1}/2^{i-1} \rfloor) + O(n_j)$ time, where $\hat{i} = \lfloor \lg N_{j-1} \rfloor$. Mathematically, $T(j, N_j) \leq \sum_{i=1}^{\hat{i}} T'(j, \lfloor N_{j-1}/2^{i-1} \rfloor) + O(n_j)$.

Second, consider Algorithm 2. To find the splitting node of the internal nodes at level η_j, we find the splitting node of all the nodes at level η_{j-1} for the same list of weights. We also find the t-th smallest and largest nodes among each half of this list of weights; the time for each of these two calls is at most $T(j-1, \lfloor |\mathcal{L}|/2 \rfloor)$. Mathematically, $T'(j, |\mathcal{L}|) \leq T(j-1, |\mathcal{L}|) + 2\, T(j-1, \lfloor |\mathcal{L}|/2 \rfloor) + O(1)$.

Summing up the bounds, the next relations follow:

$$\begin{aligned} T(1, N_1) &= O(n_1), \\ T(j, 1) &= O(1), \\ T(j, N_j) &\leq \sum_{i=1}^{\hat{i}} T(j-1, \lfloor N_{j-1}/2^{i-1} \rfloor) + 2 \sum_{i=1}^{\hat{i}-1} T(j-1, \lfloor N_{j-1}/2^{i} \rfloor) + O(n_j). \end{aligned}$$

Substitute with $T(a, b) < c \cdot 4^a \cdot b$, for $1 < a < j$, $1 < b < N_j$, and a big enough constant c. Then, we induce for $j \geq 2$ that

$$\begin{aligned} T(j, N_j) &< c \cdot 4^{j-1} \cdot N_{j-1} \Big(\sum_{i=1}^{\infty} 1/2^{i-1} + 2 \sum_{i=1}^{\infty} 1/2^{i} \Big) + O(n_j) \\ &< c \cdot 4^{j} \cdot N_{j-1} + c \cdot n_j. \end{aligned}$$

Using the fact that $N_j = N_{j-1} + n_j$, then

$$T(j, N_j) = O(4^j \cdot N_j).$$

Consider the case when the list of weights W is presorted. Let $C_s(j, N_j)$ be the number of comparisons required to find the splitting node. The number of comparisons, in all recursive calls, performed against the medians among the n_j weights assigned to level η_j, is at most $3 \lg (n_j + 1)$ (at most $\lg (n_j + 1)$ comparisons to find χ_j, another $\lg (n_j + 1)$ to find the β-th largest node among the nodes smaller than χ_j, and a third $\lg (n_j + 1)$ to find the $(\lambda - \beta - 1)$-th smallest node among those larger than χ_j). The next relations follow:

$$\begin{aligned} C_s(1, N_1) &= 0, \\ C_s(j, 1) &= 0, \\ C_s(j, N_j) &\leq \sum_{i=1}^{\hat{i}} C_s(j-1, \lfloor N_{j-1}/2^{i-1} \rfloor) + 2 \sum_{i=1}^{\hat{i}-1} C_s(j-1, \lfloor N_{j-1}/2^{i} \rfloor) + 3 \lg (n_j + 1). \end{aligned}$$

The $O(4^j \cdot N_j)$ bound, we showed for the running time of the unsorted case, obviously fulfills the preceding recursive relations as well. However, we next deduce another bound for the sorted case that is tighter when j is sufficiently small.

Since the number of recursive calls at level η_j is at most $3\hat{i} - 1 < 3 \lg N_{j-1}$, it follows that

$$\begin{aligned} C_s(j, N_j) &< 3 \lg N_{j-1} \cdot C_s(j-1, N_{j-1}) + 3 \lg (n_j + 1) \\ &< 3 \lg N_j \cdot C_s(j-1, N_{j-1}) + 3 \lg N_j. \end{aligned}$$

Substitute with $C_s(a, b) \leq 3^{a-1} \cdot \sum_{i=1}^{a-1} \lg^i b$, for $1 < a < j$, $1 < b < N_j$. We thus obtain for $j \geq 2$ that

$$\begin{aligned} C_s(j, N_j) &< 3 \lg N_j \cdot 3^{j-2} \cdot \sum_{i=1}^{j-2} \lg^i N_j + 3 \lg N_j \\ &\leq 3^{j-1} \cdot \sum_{i=1}^{j-1} \lg^i N_j \\ &= O(3^j \cdot \log^{j-1} N_j). \end{aligned}$$

Third, consider Algorithm 3. The time required by this procedure is dominated by the $O(n)$ time to find the minimum weight w among the weights remaining in W plus the time for the calls to find the splitting nodes. Let $T''(j, N_j)$ be the time required by this procedure, and let $\hat{i} = \lfloor \lg N_j \rfloor$. Then,

$$T''(j, N_j) \leq \sum_{i=1}^{\hat{i}} T(j, \lfloor N_j / 2^{i-1} \rfloor) + O(n) = O(4^j \cdot N_j + n).$$

Let $C_s''(j, N_j)$ be the number of comparisons required by Algorithm 3 when the list of weights W is presorted. Then,

$$C_s''(j, N_j) \leq \sum_{i=1}^{\hat{i}} C_s(j, \lfloor N_j / 2^{i-1} \rfloor) + O(1) = O(3^j \cdot \log^{j-1} N_j).$$

Finally, consider Algorithm 4. The required time is dominated by the time to find the weights contributing to the ν nodes with the largest ranks at level η_j, which is $O(4^j \cdot N_j)$. If W is presorted, the number of comparisons involved is $O(3^j \cdot \log^{j-1} N_j)$.

Using the bounds deduced for the described steps of the algorithm, we conclude that the time required by the general iteration is $O(4^j \cdot N_j + n)$. If W is presorted, the required number of comparisons is $O(3^j \cdot \log^{j-1} N_j)$.

To complete the analysis, we need to show the effect of maintaining Kraft inequality on the complexity of the algorithm. Consider the scenario when, as a result of moving subtrees one level up, all the weights at a level move up to the next level

that already had other weights. As a result, the number of levels that contain leaves decreases. It is possible that within a single iteration the number of such levels decreases to half its value. If this happens for several iterations, the amount of work done by the algorithm would have been significantly large compared to, k, the actual number of distinct codeword lengths. Fortunately, this scenario will not happen quite often. In the next lemma, we bound the number of iterations performed by the algorithm by $2k$. We also show that at any step of the algorithm the number of levels that are assigned weights, and hence the number of iterations performed, is at most twice the number of the distinct optimal codeword lengths for the weights that have been assigned so far. The proof of the lemma is deferred to the appendix.

Lemma 2 *Consider the set of weights that will have the τ-th largest optimal codeword length at the end of the algorithm. During the execution of the algorithm, these weights will be assigned to at most two consecutive (with respect to the levels that contain leaves) levels, with level numbers, at most, $2\tau - 1$ and 2τ. Hence, the number of iterations performed by the algorithm is at most $2k$.*

Using Lemma 2, the time required by our algorithm to assign the set of weights whose optimal codeword length is the j-th largest, among all distinct lengths, is $O(4^{2j} \cdot n) = O(16^j \cdot n)$. Summing for all such lengths, the total time required by our algorithm is $\sum_{j=1}^{k} O(16^j \cdot n) = O(16^k \cdot n)$.

Consider the case when the list of weights W is presorted. For achieving the claimed bounds, the only point left to be mentioned is how to find the weights of W smaller than the sum of the values of the smallest two nodes at level η_j. Once this sum is evaluated, we apply an exponential search that is followed by a binary search on the weights of W; this requires $O(\log n_j)$ comparisons. Using Lemma 2, the number of comparisons performed to assign the weights whose codeword length is the j-th largest among all distinct lengths is $O(9^j \cdot \log^{2j-1} n)$. Summing for all such lengths, the number of comparisons performed by our algorithm is $\sum_{j=1}^{k} O(9^j \cdot \log^{2j-1} n) = O(9^k \cdot \log^{2k-1} n)$. Our main theorem follows.

Theorem 1 *Constructing a minimum-redundancy prefix code for a set of n weights can be done in $O(16^k \cdot n)$ time, where k is the number of distinct codeword lengths. If the list of weights is presorted, the algorithm uses $O(9^k \cdot \log^{2k-1} n)$ comparisons.*

Corollary 1 *For $k < c \cdot \lg \lg n$ and any constant $c < 1/4$, the above algorithm requires $o(n \log n)$ time. If the list of weights was presorted, for $k < c \cdot \lg n / \lg \lg n^3$ and any constant $c < 1/2$, the above algorithm requires $o(n)$ comparisons*

5 Conclusion

We gave an output-sensitive algorithm for constructing minimum-redundancy prefix codes whose running time is $O(16^k \cdot n)$. For sufficiently small values of k, this algorithm asymptotically improves over other known algorithms that require $O(n \log n)$;

it is quite interesting to know that the construction of optimal codes can be done in linear time when k turns out to be a constant. For sufficiently small values of k, if the sequence of weights was presorted, the number of comparisons performed by our algorithm is asymptotically better than other known algorithms that require $O(n)$ comparisons. For such sorted sequences, the number of comparisons required by our algorithm is poly-logarithmic when k is a constant.

Two remaining questions are whether we can improve the bounds to be polynomial with respect to k, and whether it is possible to make the algorithm practically more efficient by avoiding so many recursive calls to a median-finding algorithm.

Appendix
Proof of Lemma 2

Consider a set of weights that will turn out to have the same codeword length. During the execution of the algorithm, assume that some of these weights are assigned to three levels. Let $\eta_j < \eta_{j+1} < \eta_{j+2}$ be such levels. Since we are maintaining the exclusion property throughout the algorithm and since $\eta_j + 1 < \eta_{j+2}$, there will exist some internal nodes at level $\eta_j + 1$ whose values are strictly smaller than the values of the weights at level η_{j+2} (some may have the same value as the smallest weight at level η_{j+2}). The only way for such weights to catch each other at the same tree level would be as a result of moving subtrees up to maintain Kraft inequality. Suppose that, at some point of the algorithm, the weights that are currently at level η_j are moved up to catch the weights at level η_{j+2}. It follows that the internal nodes that are currently at level $\eta_j + 1$ will accordingly move to the next upper level of the moved weights. As a result, the exclusion property will not hold; a fact that contradicts the behavior of our algorithm. It follows that these weights will never be at the same tree level.

We prove the second part of the lemma by induction. The base case follows easily for $\tau = 1$. Assume that the argument is true for $\tau - 1$. By induction, the levels of the weights that will have the $(\tau - 1)$-th largest optimal codeword length will be assigned to the at most $2\tau - 3$ and $2\tau - 2$ levels. From the exclusion property, it follows that the weights that have the τ-th largest optimal codeword length must be at the next upper levels. Using the first part of the lemma, the number of such levels is at most two. It follows that these weights are assigned to the, at most, $2\tau - 1$ and 2τ levels among those assigned weights.

Hence, the weights with the τ-th largest optimal codeword length will be assigned within 2τ iterations. Since the number of distinct codeword lengths is k, the number of iterations performed by the algorithm is at most $2k$.

References

[1] A. Belal and A. Elmasry. *Verification of minimum-redundancy prefix codes.* IEEE Transactions on Information Theory, 52(4) (2006), 1399-1404.

[2] A. Belal and A. Elmasry. *Distribution-sensitive construction of minimum-redundancy prefix codes.* 23rd Annual Symposium on Theoretical Aspects of Computer Science (2006), 92-103.

[3] M. Buro. *On the maximum length of Huffman codes.* Information Processing Letters 45 (1993), 219-223.

[4] T. Chan. *Optimal output-sensitive convex hull algorithms in two and three dimensions.* Discrete and Computational Geometry 16(4), (1996), 361-368.

[5] T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to algorithms,* 3rd Edition. The MIT press (2009).

[6] R. Gallager. *Variations on a theme by Huffman.* IEEE Transactions on Information Theory 24(6) (1978), 668-674.

[7] D. Huffman. *A method for the construction of minimum-redundancy codes.* Proc. IRE 40 (1952), 1098-1101.

[8] R. Milidiu, A. Pessoa, and E. Laber. *Three space-economical algorithms for calculating minimum-redundancy prefix codes.* IEEE Transactions on Information Theory 47(6) (2001), 2185-2198.

[9] J. I. Munro and P. Spira. *Sorting and searching in multisets.* SIAM Journal on Computing 5(1) (1976), 1-8.

[10] A. Moffat and J. Katajainen. *In-Place calculation of minimum-redundancy codes.* 4th Workshop on Algorithms and Data Structures (1995), 393-402.

[11] A. Moffat and A. Turpin. *Efficient construction of minimum-redundancy codes for large alphabets.* IEEE Transactions on Information Theory 44(4) (1998), 1650-1657.

[12] S. Sen and N. Gupta. *Distribution-sensitive algorithms.* Nordic Journal of Computing 6(2) (1999), 194-211.

[13] J. Van Leeuwen. *On the construction of Huffman trees.* 3rd International Colloquium for Automata, Languages and Programming (1976), 382-410.

[14] J. S. Vitter. *Design and analysis of dynamic Huffman codes.* Journal of the ACM 34(4) (1987), 825-845.

[15] J. Zobel and A. Moffat. *Adding compression to a full-text retrieval system.* Software: Practice and Experience 25(8) (1995), 891-903.