Alphabetic minimax trees in linear time*

Paweł Gawrychowski

Institute of Computer Science,
University of Wrocław,
ul. Joliot-Curie 15, 50–383 Wroclaw, Poland
gawry@cs.uni.wroc.pl

Abstract. We develop a linear time algorithm algorithm for the following problem: given an ordered sequence of n real weights $w_1, w_2, \ldots, w_n$, construct a binary tree on n leaves labeled with those weights when read from left to right so that the maximum value of w_i plus depth of the i-th leftmost leaf is minimized. This improves the previously known $O(n \log n)$ time solutions [2,11,13].
We also give a simplified $\mathcal{O}(nd)$ version of the algorithm, where d is the number of distinct integer parts of w_i, which does not require the full power of the word RAM model. This improves the previously known $\mathcal{O}(nd \log \log n)$ solution of Gagie [5].
Key-words: minimax tree, Yeung's inequality, word RAM

1 Introduction

Trees are one of the most common and useful objects appearing in such areas as graph theory, data structures, and information theory, to name just a few. There exists an enormous amount of research devoted to investigating various aspects of constructing *optimal* trees, for different definitions of optimality. In particular, a lot of effort has been put into solving variants of the following problem: given a collection of n weights construct a (binary, ternary, or t-ary) tree with those weights stored in the leaves so that the sum (or the maximum) of all root-to-leaf paths weights is minimized, where the path weight depends only on its length and the leaf weight.

For the case when the path weight is simply the product of those two quantities, a well-known $\mathcal{O}(n \log n)$ time solution was given by Huffman [12]. If we additionally require that the collection of weights is ordered, and they must be assigned in the natural left-to-right order in the tree, we call the problem *alphabetic*. A $\mathcal{O}(n \log n)$ time solution for the alphabetic version of the problem considered by Huffman was developed by Hu and Tucker [11]. While the solution complexity is the same in both cases, the latter is much more complicated.

For the case when we minimize the maximum path weight, and the path weight is the sum of its length and the leaf weight, Golumbic [8] modified the Huffman's algorithm to find the optimum tree, which has been then used [3,9] to

* Supported by MNiSW grant number N N206 492638, 2010–2012

restrict the fan-in and fan-out of a circuit without increasing its size too much. Recently a linear time solution in the word RAM model of computation for the problem was given [7]. Also the alphabetic version of this case has been considered before. Hu, Kleitman and Tamaki [10] observed that a certain modification of the Hu-Tucker algorithm can be used to compute the ordered minimax cost in $\mathcal{O}(n \log n)$ time (actually, their algorithm minimizes $w_i 2^{l_i}$, but this is easily seen to be equivalent). Then Kirkpatrick and Klawe [13] considered the strict t-ary version and applied their $\mathcal{O}(n \log n)$ time solution to study the effects of fan-out constraint in planar logical circuits. Later Coppersmith, Klawe and Pippinger [2] removed the strictness conditions with the same complexity. If the weights are integer, a linear time solution is known [4], and if the number of different rounded down weights is bounded by d, running time of $\mathcal{O}(nd \log \log n)$ is possible [6]. This suggest the following natural question: can a linear time solution for the alphabetic case be achieved? Or maybe when we minimize the maximum instead of the sum this additional requirement makes the problem more complex in terms of the best running time possible? In this paper we show that this is not the case. A (very) high level idea of our algorithm is the same as in the non-alphabetic case, but we need to apply a significantly more complicated reasoning in order to deal with the alphabetic constraint. Nevertheless, we are able to achieve a linear running time, assuming the word RAM model of computation.

We start with a simple linear time algorithm for the case when all w_i are integer. While this is not a new result, the characterization of ordered binary search trees we use there can be applied to develop an $\mathcal{O}(nd)$ time algorithm for the more general case when w_i are arbitrary real numbers, with d being the cardinality of $\{\lfloor w_i \rfloor : i = 1, 2, \ldots, n\}$. Then we use the power of the word RAM model more extensively in order to improve the complexity to linear.

2 Preliminaries

Given a sequence of n real weights $w_1, w_2, \ldots, w_n$, we want to construct an ordered binary search tree on n leaves labeled with those weights. The labeling is ordered: the leftmost leaf should correspond to w_1, the second leftmost to w_2, and so on. Our goal is to minimize the maximum value of w_i plus depth of the i-th leftmost leaf. This quantity will be called the ordered minimax cost $M(w_1, w_2, \ldots, w_n)$.

The algorithm we are going to present works in the word RAM model, meaning that we are allowed to perform operations on integers consisting of $\log n$ bits in constant time. We do not assume any specific encoding of the real numbers given in the input, but we do require that their representation allows performing a few basic arithmetic operations in constant time. By basic operations we mean comparing, subtracting, extracting the fractional part, and rounding to an integer if its value does not exceed n. Assuming that we can perform those operations in constant time seems natural.

We are going to work with ordered binary trees, meaning that each node can have a left and a right child. The *shape* of such tree is a sequence of n integers, the depths of successive leaves when read from left to right. We need an efficient way of checking whether a given shape corresponds to at least one tree. When the tree is not meant to be ordered, this is possible thanks to the well-known Kraft's inequality. In the ordered case the inequality is no longer useful, though. We use a different and not so well-known characterization instead.

Theorem 1 (Yeung's inequality [14]). *$\langle l_1, l_2, \ldots, l_n \rangle$ is a shape of some ordered binary tree if and only if*

$$f_{l_1} \circ f_{l_2} \circ \ldots \circ f_{l_{n-1}} \circ f_{l_n}(0) \leq 1 \tag{1}$$

where $f_a(x) = \frac{\lceil x 2^a \rceil + 1}{2^a}$.

This inequality can be viewed as a different way of saying that to construct a tree of a given shape, one can use a simple greedy method: add leaves from left to right, putting each of them as deeply as it is possible. A n-tuple $\langle l_1, l_2, \ldots, l_n \rangle$ for which the above lemma holds will be called a *valid shape*.

Having the above lemma, we can formulate the problem of computing the ordered minimax cost $M(w_1, w_2, \ldots, w_n)$ as follows: minimize $\max_i w_i + l_i$ among all $l_1, l_2, \ldots, l_n$ such that $\langle l_1, l_2, \ldots, l_n \rangle$ is a valid shape.

3 Linear time algorithm for integer weights

We begin with a rather simple linear time algorithm for the case when all w_i are integers. While it neither improves or simplifies already known solutions, it does help to understand the general real weight case algorithm.

We begin with modifying the formulation of Lemma 1 to make it more convenient to use. First of all, we do not want to use fractions. Define $g_a(x) = \left\lceil \frac{x}{2^a} \right\rceil 2^a + 2^a$ and observe that any tree can be rebuilt so that the depths of all leaves do not exceed n and the depth of any leaf does not increase. Thus we can rewrite (1) as

$$g_{\max(0,n-l_1)} \circ g_{\max(0,n-l_2)} \circ \ldots \circ g_{\max(0,n-l_{n-1})} \circ g_{\max(0,n-l_n)}(0) \leq 2^n \tag{2}$$

Lemma 1. *If all w_i are integers, $M(w_1, w_2, \ldots, w_n)$ can be calculated in linear time.*

Proof. First observe that the i-th leaf must be created at depth not exceeding $l_i \leq M(w_1, w_2, \ldots, w_n) - w_i$. Let $m = \max_i w_i - n$ and define:

$$A = g_{w_1-m} \circ g_{w_2-m} \circ \ldots \circ g_{w_n-m}(0)$$

Then if c is the smallest possible integer c such that $A \leq 2^c$ (or, in other words, c is A rounded up to the nearest power of 2), the ordered minimax cost is $m + c$.

To calculate A, first note that all $w_i - m$ are between 0 and n. We start the computation with $x = 0$ and apply $g_{w_1-m}, g_{w_2-m}, \ldots, g_{w_n}$ to the current x. Note

that because of the bounds on the $w_i - m$ and the structure of all g_{w_i-m}, the current x will be always an integer between 0 and $n2^n$. We store it as a sorted list L of bits set to 1, so for example if the current value is 100110, we store $[1, 2, 5]$. Computing $g_{w_i-m}(x)$ consists of two steps:

1. removing the prefix of L consisting of elements less than a,
2. adding 2^a twice or once to the current value of x, depending on whether we removed at least one element in the previous step or not, respectively.

For a detailed description of the procedure see INTEGER-ORDERED-MINIMAX in the appendix. To bound its running time we assign one credit to each of the elements of L, then the total complexity is easily seen to be linear. □

4 $\mathcal{O}(nd)$ time algorithm for real weights

To deal with the case of non-integer weights we start with some simple observations which allow us to reduce the general case to a sequence of integer instances.

Lemma 2. $\lfloor M(w_1, w_2, \ldots, w_n) \rfloor = M(\lfloor w_1 \rfloor, \lfloor w_2 \rfloor, \ldots, \lfloor w_n \rfloor)$.

Proof. Left side minimizes $\lfloor \max_i w_i + l_i \rfloor$ among all valid shapes, and right side minimizes $\max_i \lfloor w_i \rfloor + l_i$. This immediately gives the claim. □

Lemma 3. *$M(w_1, w_2, \ldots, w_n) \leq X$ if and only if $M(w'_1, w'_2, \ldots, w'_n) \leq \lfloor X \rfloor$ where w'_i is $\lfloor w_i \rfloor$ if $\mathrm{frac}(w_i) \leq \mathrm{frac}(X)$ and $\lfloor w_i \rfloor + 1$ otherwise.*

Proof. $M(w_1, w_2, \ldots, w_n) \leq X$ if and only if for some valid shape $\langle l_1, l_2, \ldots, l_n \rangle$ inequality $w_i + l_i \leq X$ holds for all i. As l_i is integer, there are two cases:

1. $\mathrm{frac}(w_i) \leq \mathrm{frac}(X)$, the inequality is equivalent to $\lfloor w_i \rfloor + l_i \leq \lfloor X \rfloor$,
2. $\mathrm{frac}(w_i) > \mathrm{frac}(X)$, the inequality is equivalent to $\lfloor w_i \rfloor + 1 + l_i \leq \lfloor X \rfloor$.

Hence the whole claim follows. □

The above lemma gives a simple $\mathcal{O}(n \log n)$ time algorithm. First use Lemma 1 and Lemma 2 to compute $\lfloor M(w_1, w_2, \ldots, w_n) \rfloor$. Then observe that the fractional part of the answer must be equal to some $\mathrm{frac}(w_i)$. We can compute it using a binary search inside which we use Lemma 1 and Lemma 3 (again) to check if a chosen $\mathrm{frac}(w_i)$ is bigger than $\mathrm{frac}(M(w_1, w_2, \ldots, w_n))$.

To accelerate the $\mathcal{O}(n \log n)$ time procedure, we must somehow reuse the information found by the successive steps of binary search. We split the whole $\{1, 2, \ldots, n\}$ into three parts L, C, and R. L contains indices i such that we already know that $\mathrm{frac}(w_i) \leq \mathrm{frac}(M(w_1, w_2, \ldots, w_n))$, R contains i for which we already know that $\mathrm{frac}(w_i) > \mathrm{frac}(M(w_1, w_2, \ldots, w_n))$, and C consists of all the remaining indices. At each step we select the median of $\{\mathrm{frac}(w_i) : i \in C\}$ and compare it with $\mathrm{frac}(M(w_1, w_2, \ldots, w_n))$ using Lemma 1 and Lemma 3. Depending on the outcome of this comparison we move half of the elements of C into L or R. Observe that at each step of the computation values of w_i with

$i \in L \cup R$ are permanently round up or down. This suggest that if a whole segment $i, i+1, i+2, \ldots, j$ belongs to $L \cup R$ already, we could try to somehow preprocess the function $g_{w'_i} \circ g_{w'_{i+1}} \circ g_{w'_{i+2}} \circ \ldots \circ g_{w'_j}$, where each w'_i is either $\max(0, \lfloor w_i \rfloor - m)$ or $\max(0, \lfloor w_i \rfloor + 1 - m)$, and apply the whole compositions at once instead of processing their elements one-by-one. Lets take a closer look at how such composition $g_{a_1,a_2,\ldots,a_k}(x) = g_{a_1} \circ g_{a_2} \circ \ldots \circ g_{a_k}(x)$ looks like.

Lemma 4. *If $a_1, a_2, \ldots, a_k$ are nonnegative integers, $g_{a_1,a_2,\ldots,a_k}(x) = \lceil \frac{x+r}{2^a} \rceil 2^a + c$, where $a = max_i\ a_i$ and r, c are of the form $\sum_{i=1}^{k} \alpha_i 2^{a_i}$ with $\alpha_i \in \{0, 1, 2\}$ for all i.*

Now if there are just d different values of $\max(0, \lfloor w_i \rfloor - m)$, any $g_{w'_i} \circ g_{w'_{i+1}} \circ \ldots \circ g_{w'_j}$ depends on just $\mathcal{O}(d \log n)$ bits. More specifically, let those rounded down values be $b_1 < b_2 < \ldots < b_d$. For any position k which does not belong to any block of the form $\{d_i, d_i + 1, \ldots, d_i + \lfloor \log 2n \rfloor\}$, the k-th bit of all $g_{w'_1} \circ g_{w'_2} \circ \ldots \circ g_{w'_i}(0)$ is set to zero. Hence while the numbers x, r, c we operate on might be as large as $n2^n$, there are just d blocks of $\mathcal{O}(\log n)$ consecutive indices which potentially contain bits set to 1. Thus any of those numbers can be actually described in just d machine words, each word storing the bits from a single block, see Figure 1. It is easy to see that given such *succinct* representation of x we can compute any $g_{w'_i}(x)$ in $\mathcal{O}(d)$ time. Similiarly, given the succinct representations of x, r, and c we can compute $\lceil \frac{x+r}{2^a} \rceil 2^a + c$ in $\mathcal{O}(d)$ time as well.

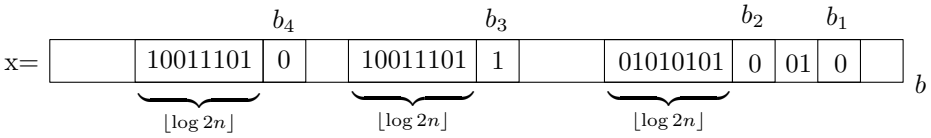

Fig. 1. Succinct representation of x is $[010_b, 010101010_b, 100111011_b, 100111010_b]$.

The last building block for the $\mathcal{O}(nd)$ time algorithm is a method for computing the description of $g_{a_1,a_2,\ldots,a_k} \circ g_{b_1,b_2,\ldots,b_\ell}$ given the descriptions of $g_{a_1,a_2,\ldots,a_k}$ and $g_{b_1,b_2,\ldots,b_\ell}$. In other words, we need to describe how a composition of two functions $h_1(x) = \left\lceil \frac{x+r_1}{2^a} \right\rceil 2^a + c_1$ and $h_2(x) = \left\lceil \frac{x+r_2}{2^b} \right\rceil 2^b + c_2$ looks like.

Lemma 5. *Let $h_1(x) = \left\lceil \frac{x+r_1}{2^a} \right\rceil 2^a + c_1$ and $h_2(x) = \left\lceil \frac{x+r_2}{2^b} \right\rceil 2^b + c_2$. If $b \le a$ then $h_2(h_1(x)) = \left\lceil \frac{x+r_1}{2^a} \right\rceil 2^a + \left\lceil \frac{c_1+r_2}{2^b} \right\rceil 2^b + c_2$, otherwise $h_2(h_1(x)) = \left\lceil \frac{x+r_1+\left\lceil \frac{c_1+r_2}{2^a} \right\rceil 2^a}{2^b} \right\rceil 2^b + c_2$.*

Using the above lemma, given the succinct representations of r_1, c_1, r_2, c_2 we can compute the representation of $h_1 \circ h_2$ in $\mathcal{O}(d)$ time. Then using such computations Slow-ordered-minimax computes $M(w_1, w_2, \ldots, w_n)$ in just $\mathcal{O}(nd)$ time, see the appendix for the details of the implementation.

Theorem 2. Slow-ordered-minimax *computes $M(w_1, w_2, \ldots, w_n)$ in $\mathcal{O}(nd)$ time, where d is the number of different values of $\lfloor w_i \rfloor$.*

Note that while we do require that the word size is $\Omega(\log n)$, we need that just a few basic arithmetic operations can be performed in constant time on such words. In order to develop a fully linear time algorithm we will need to use the word RAM model more extensively, though.

5 Linear time algorithm for real weights

At a very high level, the idea behind the fully linear time algorithm is the same as in SLOW-ORDERED-MINIMAX$(w_1, w_2, \ldots, w_n)$. We iteratively select the median fractional part of all w_i with $i \in C$ and after computing $g_{w'_1} \circ g_{w'_2} \circ \ldots \circ g_{w'_n}(0)$ remove half of C. The bottleneck is clearly the evaluation of g. Note that a running time of order $\mathcal{O}(\frac{n}{\log n})$ would be perfectably acceptable here in order to get a linear overall bound.

In order to accelerate computing g we split all indices $\{1, 2, \ldots, n\}$ into groups of consecutive $\lfloor \log n \rfloor$ elements. We call such group a *package*. Consider the function $g_{w'_i, w'_{i+1}, \ldots, w'_{i+\lfloor \log n \rfloor - 1}}$ corresponding to such package. Each w'_i is either $\max(0, \lfloor w_i \rfloor - m)$ or $\max(0, \lfloor w_i \rfloor + 1 - m)$, so by Lemma 4 this function is of the form $\lceil \frac{x+r}{2^a} \rceil 2^a + c$ with both r and c possibly containing bits set to 1 just on a specified set of $2\lfloor \log n \rfloor$ positions, no matter which w_i are rounded up and down. Let the sorted list of those positions be $[t_1, t_2, \ldots, t_{2\lfloor \log n \rfloor}]$. Note that we can easily construct all such sorted lists in linear time if we preprocess all the packages at once in the very beginning. Additionally for any index i we store the positions of bits corresponding to $\max(0, \lfloor w_i \rfloor - m)$ and $\max(0, \lfloor w_i \rfloor + 1 - m)$ on its package list. Observe that composing any sequence of $g_{w'_i} \circ g_{w'_{i+1}} \circ \ldots \circ g_{w'_j}$ with all indices belonging to the same package results in a function $\lceil \frac{x+r}{2^a} \rceil 2^a + c$ with both r and c containing bits set to 1 only on positions from the package list. This allows us to store a succinct description of such function in a constant amount of machine words. Furthermore, given such descriptions of two functions corresponding to consecutive fragments of the same package, we can compute the description of their composition in constant time by Lemma 5.

During the execution of the algorithm some w'_i are permanently rounded up or down while some are yet unknown. For each package we consider its maximal fragments consisting of indices with already known values of w'_i. We store succinct descriptions of all corresponding functions and update them accordingly whenever any w'_i becomes fixed. We claim that this allows us to construct succinct descriptions of all functions corresponding to the whole packages.

Lemma 6. *Given succinct descriptions of all functions corresponding to maximal fragments of packages consisting of indicies with already fixed values of w'_i and the current value of t, we can compute succinct descriptions of all functions corresponding to whole packages in $\mathcal{O}(|C| + \frac{n}{\log n})$ time.*

Proof. We consider the packages one by one. Consider a single package. First for any index i for which w'_i is not fixed yet, we construct a succinct description of the function $g_{\max(0, \lfloor w_p \rfloor + [\mathrm{frac}(w_p) > t] - m)}$. This requires just constant time as during

the preprocessing stage we found the position of $\max(0, \lfloor w_p \rfloor + [\mathrm{frac}(w_p) > t] - m)$ on the package list. Then we must compute a succinct description of the composition of all functions corresponding to the fragments of the current package. Due to Lemma 5 this requires time proportional to the number of those fragments. There are $\frac{n}{\log n}$ packages and $|C|$ not permanently rounded indices so the total running time is as claimed. □

Given succinct descriptions of functions corresponding to all packages, we still have to somehow evaluate their composition at 0. We would like to start with $x = 0$ and apply the functions one by one. This is not that simple to perform quickly, though. While there are just $\frac{n}{\log n}$ functions, and we already have a succinct description of each of them, lists of different packages might consists of completely different elements. Thus as a result of applying them one by one we might get x with more than just $2\lfloor \log n \rfloor$ bits set to 1, and so we cannot simply claim that applying a single function takes constant time.

By Lemma 4 as a result of applying the functions we get $0 \leq x \leq n2^n$. The obvious method of storing the current value of x would be to keep a list of its bit set to 1 as we did in Integer-ordered-minimax which would require $\Theta(n)$ time to operate on. An obvious improvement would be to store the values of $\log n$ consecutive bits in a single machine word. While it improves the storage requirements to $\mathcal{O}(\frac{n}{\log n})$, it is not clear how to apply the functions efficiently using such representation. We switch to a more complicated hybrid storage method.

Definition 1. *A hybrid representation of a n-bit integer x is an ordered list of objects. There are two types of objects:*

chunk *a description of at most $\log n$ consecutive bits,*
scattered chunk *a description of a range of bits out of which at most $\log n$ form a consecutive fragment of some package list and all the remaining are set to zero.*

For a chunk we store the values of those bits in a single machine word together with the position of the first and last bit in x. For a scattered chunk we store the values of the potentially nonzero bits in a single machine word, the corresponding package number, and positions of the first and last bit on the package list.

Note that the positions of the extreme bits on the package list in fact give us the corresponding positions in x. We call a representation valid if the ranges of bits of x referred to by the objects are disjoint and sorted. See Figure 2 for an example of such valid representation.

Consider evaluating $h(x) = \left\lceil \frac{x+r}{2^a} \right\rceil 2^a + c$ given a valid hybrid representation of x and a succinct representation of r and c. It requires performing a few steps:

1. check if the current value of x modulo 2^a plus r exceeds 2^a,
2. erase all bits on positions less than a,
3. add 2^a once or twice to the current value of x,
4. add $\left\lfloor \frac{c}{2^a} \right\rfloor 2^a$ to the current value of x,
5. add $c \bmod 2^a$ to the current value of x.

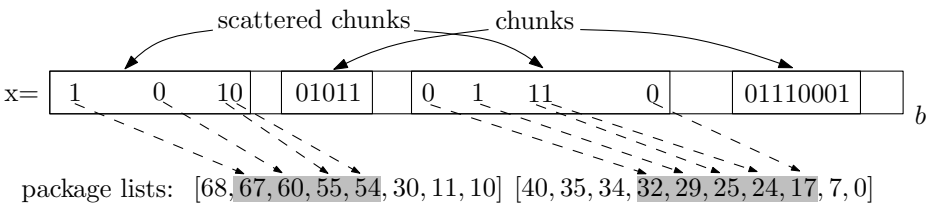

Fig. 2. Example of a valid hybrid representation.

Some of those steps are fairly simple to perform efficiently assuming we can use the hybrid representation. Consider step **5**. Succinct representation of c can be immediately converted into a scattered chunk, and after step **2** there will be no bits set to 1 at positions $0, 1, \ldots, a-1$, so this new scattered chunk can be simply appended to the current representation. Note that a is either a' or $a'+1$, where a' is the maximum value of $\max(0, w_i - m)$ with i belonging to the corresponding package. We will need some preprocessing depending on those values of a and thus we would like this a to be the same no matter how the w_i are rounded. This can be ensured by replacing step **5** with:

5(a). if $a = a'+1$ and the a'-th bit of c is set, add $2^{a'}$ to the current value of x,
5(b). add $c \bmod 2^{a'}$ to the current value of x.

Let the functions we are applying be $h_1, h_2, \ldots, h_{\frac{n}{\log n}}$. Assuming step **5** is the only place a new scatter chunk can be created, the above modification ensures the following property:

Lemma 7. *For any i and j there exists at most one package such that the i-th bit of $h_1 \circ h_2 \circ \ldots h_j(0)$ belongs to a scattered chunk referring to this package, no matter how all w_i were rounded.*

Proof. Induction on i. For i there are no scattered chunks so the claim holds. Assuming it holds for some i, applying h_{i+1} creates one new scattered chunk describing bits 0,1,...,$a'-1$ and for all higher positions the claim holds by the induction hypothesis. □

To perform all steps efficiently we need some preprocessing.

Lemma 8. *Given a scattered chunk describing bits at positions $i, i+1, \ldots, j$ we can construct a chunk describing bits at positions $i, i+1, \ldots, i+\log n-1$ and a scattered chunk describing bits at positions $i+\log n, \ldots, j$ in constant time.*

Proof. For any position k in a package list we construct construct a word with the i-th bit (with $0 \le i \le \log n$) set if and only if $i+k$ belongs to the list as well. Then given a scattered chunk we retrieve the preprocessed word. By counting the number of bits set there we calculate the shift necessary to construct the new scattered chunk. Constructing the chunk reduces to the following problem: given $b = \sum_i 2^{j_i}$ with j_i strictly increasing and $c = \sum_i \beta_i 2^i$ (both given in single words) compute the value of $\sum_i \beta_i 2^{j_i}$. This can be answered in constant time after a simple preprocessing. □

Lemma 9. *Given hybrid representations of two integers $0 \le x, y < 2^a$ we can check if $x + y \ge 2^a$ in linear time.*

Proof. To compute the carry we process the representations from right to left. Instead of performing the processing bit-by-bit, we go through whole chunks at once. The only problem is that we must be able to add two scattered chunks and a chunk to a scattered chunk efficiently. For the latter we apply Lemma 8 and simply add two chunks. For the former let the scattered chunks refer to potentially nonzero bits at positions $i_1, i_1+1, \ldots, j_1$ and $i_2, i_2+1, \ldots, j_2$, respectively. Note that if $j_1 > i_1 + 2\log n$ and $j_2 > i_2 + 2\log n$ there will be no carry no matter which bits are set. Otherwise we apply Lemma 8 twice and remove at least one scattered chunk from the representations. □

Lemma 10. *Steps* **1** *and* **2** *can be performed in amortized constant time.*

Proof. We apply Lemma 9 with $x \bmod 2^a$ and r. The hybrid representation of $x \bmod 2^a$ is a suffix of the current representation of x, with the exception that we might need to split a chunk (which is simple to perform in constant time) or a scattered chunk into two. To perform the latter in constant time, note that due to Lemma 7 we can preprocess the predecessor a' on the package list corresponding to this scattered chunk (more specifically, the list is the same no matter how we round the w_i). Using such preprocessing we can locate the predecessor of a and erase all bits on its right. This allows us to construct hybrid representations of $x \bmod 2^a$ and $\left\lfloor \frac{x}{2^a} \right\rfloor 2^a$ in time proportional to the size of the former. After applying Lemma 9 we replace x with $\left\lfloor \frac{x}{2^a} \right\rfloor 2^a$ so the time can be amortized by the decrease in the size of the representation. □

Lemma 11. *Given $0 \le t \le n$ and a hybrid representation of x we can add t to x in amortized constant time.*

Proof. We convert t into a chunk and go through the representation of x from right to left using Lemma 8 as long as there is a carry. Note that we stop as soon as we encounter a scattered chunk referring to a sufficiently large range of bits, namely at least $2\log n$. All smaller scattered chunks end up converted to chunks. To amortize to constant time we assign one credit to each chunk and two credits to each scattered chunk. □

Lemma 12. *Steps* **3** *and* **4** *can be performed in amortized constant time.*

Proof. After step **2** there are no bits at positions $0, 1, \ldots, a-1$ and thus while both 2^a and $\left\lfloor \frac{c}{2^a} \right\rfloor 2^a$ can be much larger than n, we will try to apply Lemma 11 as if the numbers were divided by 2^a. It is easy to see that we can do that with 2^a but $\left\lfloor \frac{c}{2^a} \right\rfloor 2^a$ requires more attention. By Lemma 4, $\left\lfloor \frac{c}{2^a} \right\rfloor \le \lfloor \log n \rfloor$. Unfortunately, c is given in a succinct representation, and we would like to compute $\left\lfloor \frac{c}{2^a} \right\rfloor$ as an integer. This can be done by preprocessing the position of a on the corresponding package list (note that we actually have to preprocess the positions of a' and $a'+1$ which are the possible values of a), which allows us to construct a scattered chunk describing $\left\lfloor \frac{c}{2^a} \right\rfloor 2^a$ with a constant number of bitwise operations. Then we apply Lemma 8 to compute $\left\lfloor \frac{c}{2^a} \right\rfloor$, and Lemma 11 to add it to the current x. □

Lemma 13. *Step* **5** *can be performed in constant time.*

Proof. After precomputing the position of a' on the corresponding package list we can check if the a'-th bit is set in c in constant time. If so, we append a new chunk containing just one bit at the a'-th position to the hybrid representation of x. Then we construct and append a new scattered chunk describing $c \bmod 2^{a'}$ by simply erasing bits at higher position from the succinct representation of c and converting it into the scattered chunk in constant time. □

Finally we combine Lemma 10, Lemma 12 and Lemma 13.

Theorem 3. *$M(w_1, w_2, \ldots, w_n)$ can be calculated in $\mathcal{O}(n)$ time.*

References

1. M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.
2. D. Coppersmith, M. Klawe, and N. Pippenger. Alphabetic Minimax Trees of Degree at Most t. *SIAM Journal on Computing*, 15:189, 1986.
3. P. DS Jr. Combinatorial Merging and Huffman's Algorithm. *IEEE Transactions on Computers*, pages 365–367, 1979.
4. W. Evans and D. Kirkpatrick. Restructuring ordered binary trees. *J. Algorithms*, 50:168–193, February 2004.
5. T. Gagie. A new algorithm for building alphabetic minimax trees. *Fundamenta Informaticae*, 97(3):321–329, 2009.
6. T. Gagie. A new algorithm for building alphabetic minimax trees. *Fundam. Inf.*, 97:321–329, August 2009.
7. P. Gawrychowski and T. Gagie. Minimax trees in linear time with applications. *Combinatorial Algorithms*, pages 278–288, 2009.
8. M. C. Golumbic. Combinatorial merging. *IEEE Trans. Comput.*, 25:1164–1167, November 1976.
9. H. J. Hoover, M. M. Klawe, and N. J. Pippenger. Bounding fan-out in logical networks. *J. ACM*, 31:13–18, January 1984.
10. T. Hu, D. Kleitman, and J. Tamaki. Binary trees optimum under various criteria. *SIAM Journal on Applied Mathematics*, 37(2):246–256, 1979.
11. T. Hu and A. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.
12. D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
13. D. Kirkpatrick and M. Klawe. Alphabetic minimax trees. *SIAM Journal on Computing*, 14:514, 1985.
14. R. Yeung. Alphabetic codes revisited. *Information Theory, IEEE Transactions on*, 37(3):564–572, 2002.

Algorithm 1 INTEGER-ORDERED-MINIMAX$(w_1, w_2, \ldots, w_n)$

```
1: m ← max_i w_i − n
2: L ← []
3: for i ← 1 to n do
4:     a ← max(0, w_i − m)
5:     t ← 1
6:     while L = [b, ...] and b < a do
7:         remove b from L
8:         t ← 2
9:     end while
10:    for k ← 1 to t do
11:        b ← a
12:        while L = [b, ...] do
13:            remove b from L
14:            b ← b + 1
15:        end while
16:        prepend b to L
17:    end for
18: end for
19: c ← last element of L
20: return m + c
```

Lemma 4. *If $a_1, a_2, \ldots, a_k$ are nonnegative integers, $g_{a_1,a_2,\ldots,a_k}(x) = \lceil \frac{x+r}{2^a} \rceil 2^a + c$, where $a = max_i\ a_i$ and r, c are of the form $\sum_{i=1}^{k} \alpha_i 2^{a_i}$ with $\alpha_i \in \{0, 1, 2\}$ for all i.*

Proof. Induction on k:

1. $k = 1$, obvious.
2. $k > 0$, let $b = a_k$, from the induction hypothesis:

$$g_{a_1,a_2\ldots,a_k}(x) = \left\lceil \frac{\left\lceil \frac{x+r}{2^a} \right\rceil 2^a + c}{2^b} \right\rceil 2^b + 2^b =$$

$$= \begin{cases} b \le a : \left\lceil \left\lceil \frac{x+r}{2^a} \right\rceil 2^{a-b} + \frac{c}{2^b} \right\rceil 2^b + 2^b = \left\lceil \frac{x+r}{2^a} \right\rceil 2^a + \left\lceil \frac{c}{2^b} \right\rceil 2^b + 2^b \\ b > a : \left\lceil \frac{\left\lceil \frac{x+r}{2^a} \right\rceil}{2^{b-a}} + \frac{c}{2^b} \right\rceil 2^b + 2^b = \left\lceil \frac{\left\lceil \frac{x+r}{2^a} \right\rceil + \left\lceil \frac{c}{2^a} \right\rceil}{2^{b-a}} \right\rceil 2^b + 2^b = \\ \qquad = \left\lceil \frac{x+r+\left\lceil \frac{c}{2^a} \right\rceil 2^a}{2^b} \right\rceil 2^b + 2^b \end{cases}$$

Proving that both r and c are of claimed form requires a simple inspection of the above induction. □

Lemma 5. *Let $h_1(x) = \left\lceil \frac{x+r_1}{2^a} \right\rceil 2^a + c_1$ and $h_2(x) = \left\lceil \frac{x+r_2}{2^b} \right\rceil 2^b + c_2$. If $b \le a$ then $h_2(h_1(x)) = \left\lceil \frac{x+r_1}{2^a} \right\rceil 2^a + \left\lceil \frac{c_1+r_2}{2^b} \right\rceil 2^b + c_2$, otherwise $h_2(h_1(x)) = \left\lceil \frac{x+r_1+\left\lceil \frac{c_1+r_2}{2^a} \right\rceil 2^a}{2^b} \right\rceil 2^b + c_2$.*

Proof. First consider the case of $b \leq a$:

$$h_2(h_1(x)) = \left\lceil \frac{\left\lceil \frac{x+r_1}{2^a} \right\rceil 2^a + c_1 + r_2}{2^b} \right\rceil 2^b + c_2 = \left\lceil \frac{x+r_1}{2^a} \right\rceil 2^a + \left\lceil \frac{c_1+r_2}{2^b} \right\rceil 2^b + c_2$$

and for $b > a$:

$$h_2(h_1(x)) = \left\lceil \frac{\left\lceil \frac{x+r_1}{2^a} \right\rceil 2^a + c_1 + r_2}{2^b} \right\rceil 2^b + c_2 = \left\lceil \frac{\left\lceil \frac{x+r_1}{2^a} \right\rceil 2^a + \left\lceil \frac{c_1+r_2}{2^a} \right\rceil 2^a}{2^b} \right\rceil 2^b + c_2 =$$
$$= \left\lceil \frac{x + r_1 + \left\lceil \frac{c_1+r_2}{2^a} \right\rceil 2^a}{2^b} \right\rceil 2^b + c_2$$

hence after a simple arithmetic manipulation the claim follows. □

Theorem 2. Slow-ordered-minimax *computes* $M(w_1, w_2, \ldots, w_n)$ *in* $\mathcal{O}(nd)$ *time, where* d *is the number of different values of* $\lfloor w_i \rfloor$.

Proof. We start with computing $M(\lfloor w_1 \rfloor, \lfloor w_2 \rfloor, \ldots, \lfloor w_n \rfloor)$. Initially we do not know whether w_i should be rounded up or down for any i. During the execution of Slow-ordered-minimax we repeatedly choose the median of all $\mathrm{frac}(w_i)$ with i belonging to the set of indices which we do not know how to round yet. By a well-known result the selection can be performed in $\mathcal{O}(|C|)$ time [1]. Then we temporarily round all w_i with $\mathrm{frac}(w_i)$ not exceeding this median down, and all remaining w_i up, and compute the ordered minimax cost for the rounded weights w'_i. If $M(w'_1, w'_2, \ldots, w'_n) = M(\lfloor w_1 \rfloor, \lfloor w_2 \rfloor, \ldots, \lfloor w_n \rfloor)$ we permanently move all indices corresponding to the rounded down w_i from C to L. Otherwise we move all indices corresponding to the rounded up w_i from C to R. In either case, w_i corresponding to the transferred indices stay rounded till the end of the procedure.

To evaluate $M(w'_1, w'_2, \ldots, w'_n)$ efficiently, for each maximal segment of indices $i, i+1, \ldots, j$ belonging $L \cup R$ we store a succinct description of the corresponding function $g_{w'_i, w'_{i+1}, \ldots, w'_j}$. More precisely, we store an ordered collection S of maximal segments $[\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_s, r_s]$ consisting of indices from $L \cup R$, and for each such segment we keep the corresponding function. Then computing $M(w'_1, w'_2, \ldots, w'_n)$ reduces to evaluating a composition of $s + |C|$ functions given by their succinct descriptions. Note that $s \leq |C| + 1$ so this can be performed in $\mathcal{O}(d|C|)$ time. After computing $M(w'_1, w'_2, \ldots, w'_n)$ we shrink C by moving half of its elements to either L or R. As a consequence we must update S by first adding singleton segments $[i, i]$ for all i removed from C and then gluing together all pairs of neighboring segments of the form $[\ell_i, r_i]$, $[\ell_{i+1}, r_{i+1}]$. Both operations require just $\mathcal{O}(|C|)$ operations on succinct descriptions of either integers or functions.

The total running time is $\mathcal{O}(d(n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots)) = \mathcal{O}(nd)$. □

Algorithm 2 SLOW-ORDERED-MINIMAX$(w_1, w_2, \ldots, w_n)$

$I \leftarrow M(\lfloor w_1 \rfloor, \lfloor w_2 \rfloor, \ldots, \lfloor w_n \rfloor)$ ▷ **Lemma 1**
$m \leftarrow \max_i \lfloor w_i \rfloor - n$
$L, R \leftarrow \emptyset$
$C \leftarrow \{1, 2, \ldots, n\}$
$S \leftarrow []$
while $|C| > 0$ **do**
  $t \leftarrow$ median of $\{\mathrm{frac}(w_i) : i \in C\}$
  $p \leftarrow 1$
  let $S = [\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_s, r_s]$ and $\ell_{s+1} = n + 1$
  $x \leftarrow 0$
  for $i \leftarrow 1$ to $s + 1$ **do**
    while $p < \ell_i$ **do**
      $x \leftarrow g_{\lfloor w_p \rfloor + [\mathrm{frac}(w_p) > t] - m}(x)$
      $p \leftarrow p + 1$
    end while
    if $i \leq s$ **then**
      $x \leftarrow g_{w'_{\ell_i}, w'_{\ell_i+1}, \ldots, w'_{r_i}}(x)$ ▷ using stored succinct description
      $p \leftarrow r_i + 1$
    end if
  end for
  choose smallest c such that $x \leq 2^c$
  for all $i \in C$ s.t. $m + c = I + \left[\left(\frac{(}{w\ i}\right) > t\right]$ **do**
    remove i from C
    $w'_i \leftarrow \lfloor w_p \rfloor + \left[\left(\frac{(}{w\ p}\right) > t\right] - m$
    add $[i, i]$ to S
  end for
  merge pairs $[\ell_i, r_i]$, $[\ell_{i+1}, r_{i+1}] \in S$ with $r_i + 1 = \ell_{i+1}$
end while
return $A + \max_{i \in L} \mathrm{frac}(w_i)$