


A Data Structure for the Maximum-Sum Segment Problem with Offsets

Yoshifumi Sakai  

Graduate School of Agricultural Science, Tohoku University, Japan

Abstract

Consider a variant of the maximum-sum segment problem for a sequence X_0 of n real numbers, which asks an arbitrary contiguous subsequence of X_a that maximizes the sum of its elements for any given real number a , where X_a is the sequence obtained by subtracting a from each element in X_0 . Although this problem can be solved in $O(n)$ time from scratch for any given X_0 and a , appropriate data structures for X_0 could support efficient queries of the solution for arbitrary a . We propose an $O(n \log^2 n)$ -time, $O(n)$ -space algorithm that takes X_0 as input and outputs such a data structure supporting $O(\log n)$ -time queries.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases algorithms, sequence of real numbers, maximum-sum segment

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.26

Funding This work was supported by JSPS KAKENHI Grant Number JP23K10975.

1 Introduction

Given a sequence of real numbers, the maximum-sum segment (MSS) problem, also known as the maximum subarray problem, is to find an arbitrary segment (contiguous subsequence) of the sequence that maximizes the sum of its elements, which we call an MSS of the sequence. This problem has many applications in various industrial and academic fields such as image processing [7], pattern recognition [12], and biological sequence analysis [16]. For example, in biological sequence analysis, when the similarity between amino acids at corresponding positions in multiple amino acid sequences encoding homologous proteins is given as a score, the most highly conserved region of the sequences that is expected to play an important role [15] can be found by solving this problem [16].

The MSS problem is solvable in linear time by Kadane's algorithm, as surveyed in [2]. Due to the existence of applications in biological sequence analysis, various variants and related problems of the MSS problem have also been considered. Chen and Chao [3] designed a linear-time constructible data structure that supports constant-time queries of an MSS for any segment of the sequence. A maximal local MSS is a local MSS that is not a segment of any local MSS other than it, where a local MSS is a segment that has itself as its only MSS. Ruzzo and Tompa [13] showed that all distinct maximal local MSSs can be determined in linear time, and Sakai [14] designed a linear-time constructible data structure supporting constant-time queries of the maximal local MSS of any given segment that contains any given position. Bangtsson and Chen [1] showed that an arbitrarily given number of non-overlapping segments that maximize the sum of all their elements can be found in linear time. Yu et al. [17] considered the MSS problem where each element of the input sequence is uncertain within a specific interval and proposed a linear-time algorithm for this problem. The density of a segment is defined as the mean of all elements in the segment. Cheng et al. [4] considered the MSS problem with the condition that the density of the segment to be found is between given lower and upper bounds and showed that the problem is solvable in linearithmic time, or in linear time if we do not adopt the upper bound condition. The



© Yoshifumi Sakai;

licensed under Creative Commons License CC-BY 4.0

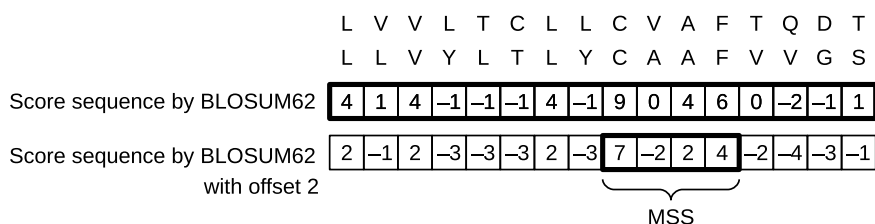
35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024).

Editors: Shunsuke Inenaga and Simon J. Puglisi; Article No. 26; pp. 26:1–26:15

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Part of an alignment of a pair of homologous amino acid sequences, the score sequence by BLOSUM62 for it, and the same score sequence with offset 2, where thick frames represent the MSSs of the score sequences.

maximum-density segment problem [8, 10, 11] consists of finding an arbitrary segment of length between given lower and upper bounds that maximizes the density. Chung and Lu [5] showed that this problem is solvable in linear time.

The present article considers another variant of the MSS problem. Before presenting the definition, we discuss the motivation that led us to conceive this new variant. As an application of the MSS problem, consider finding a highly conserved region in a given pair of homologous amino acid sequences $u_1u_2 \cdots u_n$ and $v_1v_2 \cdots v_n$ with the i th amino acids u_i and v_i appearing at the corresponding positions. For any pair of amino acids u and v , let $score(u, v)$ be the logarithm of the ratio of the observed frequency to the expected frequency of u appearing U and v appearing V at the corresponding positions over all pairs of homologous amino acid sequences U and V . From this definition, $score(u, v)$ can be regarded as representing the similarity between amino acids u and v based on the likelihood of substitution as an accepted mutation. Tables designed to consist of $score(u, v)$ approximations are available as typical amino acid substitution matrices, including PAM matrices [6] and BLOSUMs [9]. Since the larger $score(u, v)$ is, the more similar amino acids u and v are, one might think that finding an MSS of $score(u_1, v_1)score(u_2, v_2) \cdots score(u_n, v_n)$ would yield a highly conserved region with respect to $u_1u_2 \cdots u_n$ and $v_1v_2 \cdots v_n$. However, the obtained MSS may be unnecessarily large due to the low threshold level for treating amino acids u and v as sufficiently similar. To resolve this undesirable situation, we can raise the low threshold level as we wish by treating the $score(u, v)$ value as decreased by a specific value that is set as an offset. Figure 1 shows an example of how introducing such an offset changes the MSS, in which, instead of $score(u, v)$ s, values from BLOSUM62, one of BLOSUMs [9], are used. To obtain a highly conserved region as desired, it will be necessary to carefully adjust the offset, in some cases by more of a trial-and-error approach. It is possible to obtain the MSS by running Kadane's algorithm for each offset that is assumed to be appropriate. However, if a more efficient way to obtain the MSS for any given offset is available, this is the way to go.

The new variant of the MSS problem we consider is as follows. Let an offset-MSS data structure for a sequence X_0 of n real numbers be a data structure that supports queries of an arbitrary MSS of X_a for any real number a , where X_a denotes the sequence obtained from X_0 by subtracting a from each element. This type of query can arise when no firm meaning is evident in the value of each element in X_0 compared to 0 and only the relative differences in the values of elements are meaningful. This is because the MSS found can vary depending on the threshold level that separates positive from negative. For example, as mentioned earlier, in biological sequence analysis, by adjusting the criteria that separate whether each pair of amino acids is treated as similar or dissimilar, new regions may be identified as highly conserved in homologous amino acid sequences. In this article, we propose a straightforward $O(n)$ -space offset-MSS data structure for X_0 supporting $O(\log n)$ -time queries and design an $O(n \log^2 n)$ -time, $O(n)$ -space algorithm that constructs this data structure.

2 Preliminaries

Let n be an arbitrary positive integer and let X_0 be an arbitrary sequence of n real numbers. For any real number a , let X_a denote the sequence obtained by replacing each element x of X_0 with $x - a$. For any index pair (i, j) with $1 \leq i \leq j + 1 \leq n + 1$, let $X_a(i, j)$ denote the contiguous subsequence consisting of the i th through j th elements of X_a . Note that $X_a(i, j)$ is non-empty (resp. empty), if $i \leq j$ (resp. $i = j + 1$). Let $S_a(i, j)$ denote the sum of all elements in $X_a(i, j)$, if $i \leq j$, or 0, otherwise. A *maximum-sum segment* (an *MSS*) of $X_a(i, j)$ is an arbitrary index pair (g, h) with $i \leq g \leq h + 1 \leq j + 1$ that maximizes $S_a(g, h)$. We define an *offset-MSS data structure* for X_0 as a data structure that supports queries of an arbitrary MSS of X_a for any real number a . Our aim is to design an efficient algorithm that outputs an efficient offset-MSS data structure for X_0 . We assume that X_0 is given as an array of the sums $S_0(1, k)$ for all indices k with $0 \leq k \leq n$ in ascending order of k , so that $S_0(i, j)$ can be determined as $S_0(1, j) - S_0(1, i - 1)$ in $O(1)$ time.

As an efficient offset-MSS data structure for X_0 , we consider a partition of the whole set of real numbers into several intervals each with a common MSS. More precisely, our goal is to design an efficient algorithm that finds a sequence consisting of $O(n)$ pairs $(\theta, (i, j))$ of a real number θ and an index pair (i, j) with $1 \leq i \leq j + 1 \leq n + 1$ in descending order of θ such that for any real number a , if $(\theta, (i, j))$ is the last element with $\theta > a$ in the sequence, then (i, j) is an MSS of X_a . Let $OMSS_{X_0}$ denote an arbitrary such sequence. Apparently, $OMSS_{X_0}$ can be used as an offset-MSS data structure, which supports $O(\log n)$ -time queries by performing a binary search.

Below we introduce the terminology and notations used to design our algorithm. For any real number a and any index pair (i, j) with $1 \leq i \leq j + 1 \leq n + 1$, let $X_a(i, j)$ be called *pref/suff-positive*, if both $S_a(i, k)$ and $S_a(k, j)$ are positive for any index k with $i \leq k \leq j$. Let $\alpha(i, j)$ denote the least real number such that $X_{\alpha(i, j)}(i, j)$ is not pref/suff-positive, if $i \leq j$, or ∞ , otherwise. Let $\kappa(i, j)$ denote an arbitrary index k with $i \leq k \leq j$ such that at least one of $S_{\alpha(i, j)}(i, k) = 0$ or $S_{\alpha(i, j)}(k, j) = 0$, if $i \leq j$, or be undefined, otherwise. For any index pair (i, j) with $1 \leq i \leq j \leq n$, let $\delta(i, j)$ denote the real number a such that $S_a(i, j) = 0$, which is given as the *density* of $X_0(i, j)$, i.e., the mean $S_0(i, j)/(j - i + 1)$ of all elements in $X_0(i, j)$. As demonstrated in [10], it is useful to incorporate a geometric perspective when dealing with density and considering convex hulls. For any set \mathcal{P} of distinct points (p, w) in the two-dimensional plane, we define the *lower (resp. upper) convex hull* of \mathcal{P} to be the polygonal chain with the smallest number of points in \mathcal{P} as vertices such that for any point (p, w) in \mathcal{P} , there exists a pair of consecutive vertices the straight line between which passes through a point (p, w') with $w' \leq w$ (resp. $w' \geq w$).

3 Algorithm constructing an offset-MSS data structure

In this section, we show that $OMSS_{X_0}$ exists and design Algorithm `findOMSS` as an algorithm that finds $OMSS_{X_0}$ in $O(n \log^2 n)$ time and $O(n)$ space.

Algorithm `findOMSS` finds $OMSS_{X_0}$ based on a technique that divides the problem of finding an MSS into two subproblems, which is presented in the following lemma.

► **Lemma 1.** *For any real number a and any index pair (i, j) with $1 \leq i \leq j \leq n$, if $X_a(i, j)$ is pref/suff-positive, then (i, j) is the only MSS of $X_a(i, j)$; otherwise, at least one of an arbitrary MSS of $X_a(i, \kappa(i, j) - 1)$ and an arbitrary MSS of $X_a(\kappa(i, j) + 1, j)$ is an MSS of $X_a(i, j)$.*

■ **Table 1** Notations used in Section 3.1.

Notation	Definition
$\kappa'(i, j)$	An arbitrary index k with $i \leq k \leq j$ that minimizes $\delta(i, k)$
$\kappa'(i, g, h)$	An arbitrary index k with $g \leq k \leq h$ that minimizes $\delta(i, k)$
$H(g, h)$	The lower convex hull for all two-dimensional points $(k, S_0(1, k))$ with $g \leq k \leq h$
$K'(g, h)$	The sequence of all indices k with $g \leq k \leq h$ such that $(k, S_0(1, k))$ is a vertex of $H(g, h)$ in ascending order
\mathcal{K}'	The set of sequences $K'(g, h)$ for all canonical index pairs (g, h) , where (g, h) is canonical if $1 \leq g \leq h \leq n$, $h - g + 1$ is a power of two, and both $g - 1$ and h are divisible by $h - g + 1$
$k(g, h^*)$	The greatest index that is shared by $K'(g, h)$ and $K'(g, h^*)$, where (g, h) is a canonical index pair with $g < h$ and $h^* = (g + h - 1)/2$
$k(g^*, h)$	The greatest index that is shared by $K'(g, h)$ and $K'(g^*, h)$, where (g, h) is a canonical index pair with $g < h$ and $g^* = (g + h + 1)/2$
K'	The forest of binary trees such that the set of vertices consists of all canonical index pairs (g, h) and each vertex (g, h) with $g < h$ has as children (g, h^*) with label $k(g, h^*)$ and (g^*, h) with label $k(g^*, h)$, which is the $O(n)$ -time constructible data structure that supports $O(\log^2 n)$ -time queries of $\kappa'(i, j)$ for any index pair (i, j) with $1 \leq i \leq j \leq n$ we propose
K'	An implementation of K' , which is defined as the array of arrays $K'[l]$ with $0 \leq l \leq \lfloor \log_2 n \rfloor - 1$, where $K'[l]$ consists of elements $K'[l][m]$ with $1 \leq m \leq 2 \lfloor n/2^{l+1} \rfloor$, each containing the label of the vertex $(2^l(m-1)+1, 2^l m)$ of K'

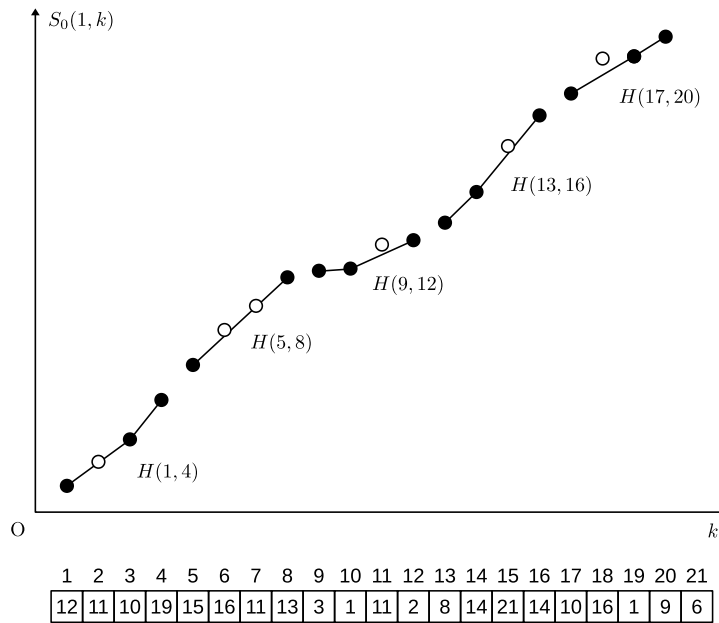
Proof. Let (g, h) be an arbitrary index pair with $i \leq g \leq h \leq j$. If $X_a(i, j)$ is pref/suff-positive and $i < g$ (resp. $h < j$), then $S_a(i, g-1)$ is positive (resp. non-negative) and $S_a(h+1, j)$ is non-negative (resp. positive), implying that $S_a(i, j) > S_a(g, h)$. Suppose that $X_a(i, j)$ is not pref/suff-positive and $g \leq \kappa(i, j) \leq h$. By symmetry, it suffices to show that if $S_{\alpha(i, j)}(i, \kappa(i, j)) = 0$, then $S_a(\kappa(i, j) + 1, h) \geq S_a(g, h)$. Since $S_{\alpha(i, j)}(i, g-1) \geq 0$ due to definition of $\alpha(i, j)$, $S_{\alpha(i, j)}(g, \kappa(i, j)) \leq 0$. Therefore, $S_{\alpha(i, j)}(\kappa(i, j) + 1, h) \geq S_{\alpha(i, j)}(g, h)$, which implies that $S_a(\kappa(i, j) + 1, h) \geq S_a(g, h)$ due to $a \geq \alpha(i, j)$. ◀

Whenever applying Lemma 1, we need $\alpha(i, j)$ to investigate whether $X_a(i, j)$ is pref/suff-positive, and $\kappa(i, j)$ if it is not. To support time-efficient queries of $\alpha(i, j)$ and $\kappa(i, j)$, one might think of a lookup table as a naive data structure, which supports $O(1)$ -time queries. However, it takes $O(n^2)$ time to construct it and also requires $O(n^2)$ space to store it. We design another data structure by taking a different approach to reduce preprocessing time and space requirement to $O(n)$ but manage to achieve $O(\log^2 n)$ -time queries.

The remaining part of this section is organized as follows. We first propose an $O(n)$ -time constructible data structure that supports $O(\log^2 n)$ -time queries of $\alpha(i, j)$ and $\kappa(i, j)$ for any index pair (i, j) with $1 \leq i \leq j \leq n$ in Section 3.1, and then design Algorithm findOMSS using this data structure in Section 3.2.

3.1 Data structure supporting queries of $\alpha(i, j)$ and $\kappa(i, j)$

The data structure we propose to support queries of $\alpha(i, j)$ and $\kappa(i, j)$ consists of two symmetric components, K' and K'' . This symmetry is based on the following reduction of the problem of determining $\alpha(i, j)$ into two symmetric subproblems. Let $\kappa'(i, j)$ (resp.



■ **Figure 2** Lower convex hulls $H(1, 4)$, $H(5, 8)$, $H(9, 12)$, $H(13, 16)$, and $H(17, 20)$ for a concrete example of X_0 shown at the bottom, where each point $(k, S_0(1, k))$ with $1 \leq k \leq 20$ is indicated by a solid bullet, if it is a vertex of the hulls, or an open bullet, otherwise.

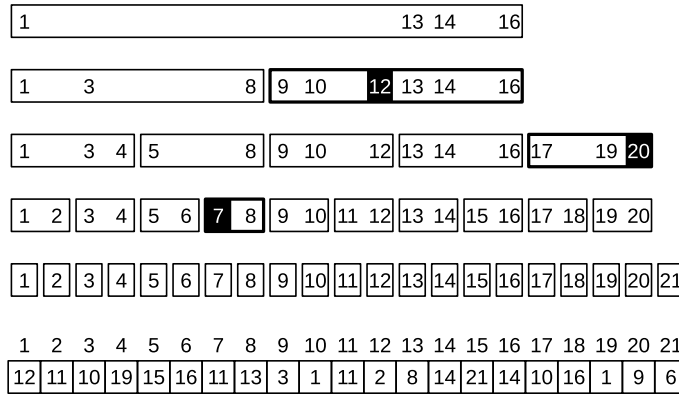
$\kappa''(i, j)$) denote an arbitrary index k with $i \leq k \leq j$ that minimizes $\delta(i, k)$ (resp. $\delta(k, j)$). Hence, $\alpha(i, j)$ is equal to the minimum of $\delta(i, \kappa'(i, j))$ and $\delta(\kappa''(i, j), j)$. Furthermore, if $\delta(i, \kappa'(i, j)) = \alpha(i, j)$ (resp. $\delta(\kappa'(i, j), j) > \alpha(i, j)$), then $\kappa'(i, j)$ (resp. $\kappa''(i, j)$) satisfies the condition of $\kappa(i, j)$. Based on this reduction, if K' supports $O(\log^2 n)$ -time queries of $\kappa'(i, j)$ and K'' supports $O(\log^2 n)$ -time queries of $\kappa''(i, j)$, then $\alpha(i, j)$ and $\kappa(i, j)$ can be determined in $O(\log^2 n)$ time. By symmetry, we will henceforth focus only on designing K' as a data structure that can be constructed in $O(n)$ time and supports $O(\log^2 n)$ -time queries of $\kappa'(i, j)$ for any index pair (i, j) with $1 \leq i \leq j \leq n$.

This section introduces many other notations besides $\kappa'(i, j)$ and K' . Table 1 summarizes such notations.

Our strategy to achieve $O(\log^2 n)$ -time queries of $\kappa'(i, j)$ is to reduce the problem of finding $\kappa'(i, j)$ to the subproblems of finding certain $O(\log n)$ candidates from which $\kappa'(i, j)$ can be found and to design a data structure that supports $O(\log n)$ -time queries of the candidate. For any index pair (g, h) with $1 \leq g \leq h \leq n$, let $K'(g, h)$ denote the sequence of all indices k with $g \leq k \leq h$ such that $(k, S_0(1, k))$ is a vertex of $H(g, h)$ in ascending order, where $H(g, h)$ denotes the lower convex hull for all two-dimensional points $(k, S_0(1, k))$ with $g \leq k \leq h$ (see Figure 2). Below is a key lemma that will serve as the foundation for our strategy.

► **Lemma 2.** For any indices i, g , and h with $1 \leq i \leq g \leq h \leq n$, a binary search of $K'(g, h)$ finds an index k with $g \leq k \leq h$ that minimizes $\delta(i, k)$.

Proof. For any index k with $g \leq k \leq h$, $\delta(i, k)$ is equal to the slope $(S_0(1, k) - S_0(1, i - 1)) / (k - (i - 1))$ of the straight line passing through points $(i - 1, S_0(1, i - 1))$ and $(k, S_0(1, k))$. Thus, the lemma follows from the fact that for any index k with $g \leq k \leq h$ that minimizes $\delta(i, k)$, the line passing through $(i, S_0(1, i - 1))$ is tangent to $H(g, h)$ at vertex $(k, S_0(1, k))$. ◀



■ **Figure 3** Set \mathcal{K}' for the same X_0 as Figure 2 shown at the bottom, where all elements in $K'(g, h)$ for each canonical index pair (g, h) are presented as indices in the rectangle lying between positions g and h and, for example, the highlighted indices represent $\kappa'(7, 7, 8)$, $\kappa'(7, 9, 16)$, and $\kappa'(7, 17, 20)$, which are obtained as candidates for determining $\kappa'(7, 20)$ ($= 12$).

A naive data structure immediately suggested by Lemma 2, which consists of sequences $K'(i, j)$ for all index pairs (i, j) with $1 \leq i \leq j \leq n$, supports $O(\log n)$ -time queries of $\kappa'(i, j)$ but consumes $O(n^3)$ space. Thus, we cannot adopt this naive data structure as is. However, by carefully choosing its particular subset, we can obtain an $O(n \log n)$ -space data structure that supports $O(\log^2 n)$ -time queries. This subset, which we denote by \mathcal{K}' , consists of sequences $K'(g, h)$ for all index pairs (g, h) with $1 \leq g \leq h \leq n$ such that $h - g + 1$ is a power of two and both $g - 1$ and h are divisible by $h - g + 1$ (see Figure 3). We call any such index pair (g, h) *canonical*. Note that \mathcal{K}' can be stored in $O(n \log n)$ space because for any power ℓ of two with $1 \leq \ell \leq n$, there exist at most n/ℓ canonical pairs (g, h) such that $h - g + 1 = \ell$, each having $K'(g, h)$ that can be stored in $O(\ell)$ space. The interval of indices represented by any index pair (i, j) with $1 \leq i \leq j \leq n$ is partitioned into $O(\log n)$ intervals each represented by a canonical index pair (g, h) . By applying Lemma 2 to each of such $O(\log n)$ canonical index pairs (g, h) , we can obtain $O(\log n)$ candidates $\kappa'(i, g, h)$ in $O(\log^2 n)$ time, where $\kappa'(i, g, h)$ denotes an arbitrary index k with $g \leq k \leq h$ that minimizes $\delta(i, k)$. Any of the candidates $\kappa'(i, g, h)$ that minimizes $\delta(i, \kappa'(i, g, h))$ satisfies the condition of $\kappa'(i, j)$.

The only reason that \mathcal{K}' cannot be adopted as K' is that the space required to store it is $O(n \log n)$, not $O(n)$. To resolve this issue, we define K' by removing duplicate information from \mathcal{K}' . We do this based on the fact that $K'(g, h)$ for any canonical index pair (g, h) with $g < h$ can recursively be represented by $K'(g, h^*)$, $K'(g^*, h)$, and two specific indices, where $h^* = (g + h - 1)/2$ and $g^* = (g + h + 1)/2$ ($= h^* + 1$). One of the specific indices is the greatest element of $K'(g, h^*)$ that is shared by $K'(g, h)$ and the other is the least element of $K'(g^*, h)$ that is shared by $K'(g, h)$. Let $k(g, h^*)$ and $k(g^*, h)$ denote these indices, respectively. Note that the concatenation of the prefix of $K'(g, h^*)$ with $k(g, h^*)$ as the last element followed by the suffix of $K'(g^*, h)$ with $k(g^*, h)$ as the first element constitutes $K'(g, h)$. This can be verified because $K'(g, h)$, $K'(g, h^*)$, and $K'(g^*, h)$ are defined to represent all vertices of the lower convex hulls $H(g, h)$, $H(g, h^*)$, and $H(g^*, h)$, respectively. For example, if X_0 is the same as Figure 2, then $K'(1, 8) = \langle 1, 3, 8 \rangle$ can be represented by $K'(1, 4) = \langle 1, 3, 4 \rangle$, $K'(5, 8) = \langle 5, 8 \rangle$, $k(1, 4) = 3$, and $k(5, 8) = 8$ in this manner. Thus, in our recursive representation, the only information that must be explicitly retained with respect to (g, h) to recover $K'(g, h)$ is indices $k(g, h^*)$ and $k(g^*, h)$.

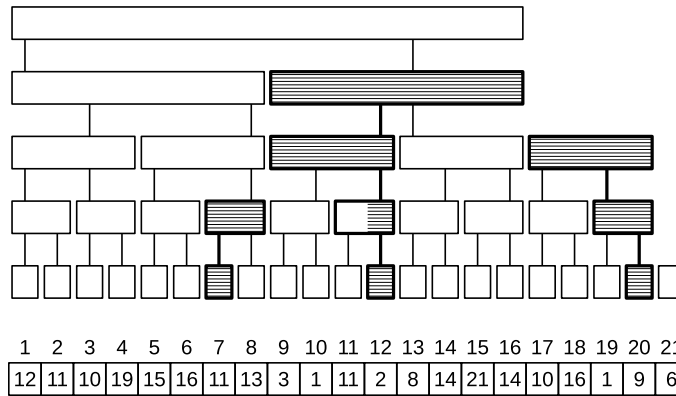


Figure 4 Forest K' for the same X_0 as Figure 2 shown at the bottom, where each rectangle lying between positions g and h indicates vertex (g, h) , the position of each edge represents its label, and the shaded areas indicate the ranges between k_{i-} and k_{i-} when tracing paths by Algorithm `determineKappa` presented in Algorithm 2 to determine $\kappa'(7, 20)$.

Based on the ideas discussed above, we define K' as a forest of binary trees as follows (see also Figure 4). The set of vertices in K' consists of all canonical index pairs (g, h) with $1 \leq g \leq h \leq n$. Each vertex (g, h) with $g < h$ has two children, (g, h^*) and (g^*, h) , where $h^* = (g + h - 1)/2$ and $g^* = (g + h + 1)/2$. Furthermore, the edge between (g, h) and (g, h^*) is labeled with $k(g, h^*)$, the greatest index in $K'(g, h^*)$ that is shared by $K'(g, h)$. Analogously, the edge between (g, h) and (g^*, h) is labeled with $k(g^*, h)$, the least index in $K'(g^*, h)$ that is shared by $K'(g, h)$. Any vertex (g, h) with $g = h$ is a leaf. Since the number of all canonical index pairs is $O(n)$, K' can be stored in $O(n)$ space, unlike the case of \mathcal{K}' , which requires $O(n \log n)$ space.

We implement K' as an array \mathbf{K}' of arrays $\mathbf{K}'[l]$ with $0 \leq l \leq \lfloor \log_2 n \rfloor - 1$, where $\mathbf{K}'[l]$ consists of elements $\mathbf{K}'[l][m]$ with $1 \leq m \leq 2 \lfloor n/2^{l+1} \rfloor$, each containing the label of the edge between vertex $(2^l(m - 1) + 1, 2^l m)$ and its parent. Algorithm 1 presents a pseudo-code of Algorithm `constructK`, which we propose as an algorithm that constructs K' in $O(n)$ time. The correctness of the algorithm and the execution time are shown in the following lemma.

► **Lemma 3.** *Algorithm `constructK` outputs K' in $O(n)$ time as an implementation of forest K' .*

Proof. For any index l with $0 \leq l \leq \lfloor \log_2 n \rfloor$, let \mathcal{K}'_l denote the array of sequences $K'(g, h)$ for all of the $\lfloor n/2^l \rfloor$ canonical index pairs (g, h) with $1 \leq g \leq h \leq n$ and $h - g + 1 = 2^l$ in ascending order of g . To construct \mathbf{K}' in $O(n)$ time, Algorithm `constructK` initializes \mathbf{K}_* to \mathcal{K}'_0 (by line 2) and updates it from \mathcal{K}'_{l-1} to \mathcal{K}'_l (by lines 4 through 15) for each index l from 1 to $\lfloor \log_2 n \rfloor$. In this process, when $K'(g, h^*)$ and $K(g^*, h)$ in \mathcal{K}'_{l-1} are merged into $K'(g, h)$ in \mathcal{K}'_l , two labels $k(g, h^*)$ and $k(g^*, h)$ are also obtained, where (g, h^*) and (g^*, h) are children of (g, h) . Let l and m be the indices such that $h = 2^l m (= 2^{l-1} \cdot 2m)$ and hence $h^* = 2^{l-1}(2m - 1)$. The loop executed by lines 5 through 13 for l and m repeatedly deletes the last element k_{-1} of the current prefix of $K'(g, h^*)$ stored in $\mathbf{K}_*[2m - 1]$, if $(k_{-1}, S_0(1, k_{-1}))$ is not a vertex of $H(g, h)$, or the first element k_1 of the current suffix of $K'(g^*, h)$ stored in $\mathbf{K}_*[2m]$, otherwise, until both $(k_{-1}, S_0(1, k_{-1}))$ and $(k_1, S_0(1, k_1))$ are vertices of $H(g, h)$. If the condition of line 8 holds, then $(k_{-1}, S_0(1, k_{-1}))$ is not a vertex of $H(g, h)$ because it is on or above the straight line passing through $(k_{-2}, S_0(1, k_{-2}))$ and $(k_1, S_0(1, k_1))$. The analogy holds for k_1 . Furthermore, if neither of the conditions in lines 8 and 10 holds,

■ **Algorithm 1** A pseudo-code of Algorithm `constructK`.

```

1  $K' \leftarrow$  an array of arrays  $K'[l]$  with  $0 \leq l \leq \lfloor \log_2 n \rfloor - 1$  each consisting of elements  $K'[l][m]$ 
   with  $1 \leq m \leq 2 \lfloor n/2^{l+1} \rfloor$ ;
2  $K_* \leftarrow$  an array of  $n$  elements  $K_*[k]$  with  $1 \leq k \leq n$  each initialized to a bidirectional linked
   list consisting of a single element  $k$ ;
3 foreach  $l$  from 1 to  $\lfloor \log_2 n \rfloor$  do
4   foreach  $m$  from 1 to  $\lfloor n/2^l \rfloor$  do
5     while not broken do
6        $k_{-1}, k_{-2} \leftarrow$  the last and second last elements of  $K_*[2m-1]$ , respectively;
7        $k_1, k_2 \leftarrow$  the first and second elements of  $K_*[2m]$ , respectively;
8       if  $k_{-2}$  exists and  $\delta(k_{-2}+1, k_{-1}) \geq \delta(k_{-2}+1, k_1)$  then
9         | delete  $k_{-1}$  from  $K_*[2m-1]$ 
10      else if  $k_2$  exists and  $\delta(k_{-1}+1, k_2) \geq \delta(k_1+1, k_2)$  then
11        | delete  $k_1$  from  $K_*[2m]$ 
12      else
13        | break
14       $K'[l-1][2m-1] \leftarrow k_{-1}$ ;  $K'[l-1][2m] \leftarrow k_1$ ;
15       $K_*[m] \leftarrow$  the concatenation of  $K_*[2m-1]$  followed by  $K_*[2m]$ 
16 output  $K'$  as an implementation of  $K'$ 

```

then both $(k_{-1}, S_0(1, k_{-1}))$ and $(k_1, S_0(1, k_1))$ are vertices of $H(g, h)$. This is because the lower convex hull for the existing $(k_{-2}, S_0(1, k_{-2}))$, $(k_{-1}, S_0(1, k_{-1}))$, $(k_1, S_0(1, k_1))$, and $(k_2, S_0(1, k_2))$ has all of them as vertices. Thus, just after the loop is broken by line 13, $k_{-1} = k(g, h^*)$ and $k_1 = k(g^*, h)$, which are respectively stored as appropriate elements of K' by line 14. In addition, the concatenation of the eventual prefix of $K'(g, h^*)$ followed by the eventual suffix of $K'(g^*, h)$ constitutes $K'(g, h)$, which is stored in $K_*[m]$ by line 15. Therefore, Algorithm `constructK` outputs K' correctly.

Each element of K_* is implemented as a bidirectional linked list and line 15 directly concatenates the lists pointed to by $K_*[2m-1]$ and $K_*[2m]$, respectively, and sets the resulting list to the one pointed to by $K_*[m]$ in $O(1)$ time. Hence, the algorithm runs in time linear in the sum of the number of canonical index pairs and the number of indices deleted by lines 9 and 11, both of which are $O(n)$. ◀

As an algorithm that allows array K' to support $O(\log^2 n)$ -time queries of $\kappa'(i, j)$, we propose Algorithm `determineKappa`(i, j) a pseudo-code of which is presented in Algorithm 2.

► **Lemma 4.** *Forest K' , implemented as array K' , supports $O(\log^2 n)$ -time queries of $\kappa'(i, j)$ for any index pair (i, j) with $1 \leq i \leq j \leq n$ by executing Algorithm `determineKappa`(i, j).*

Proof. Algorithm `determineKappa`(i, j) consists of two phases.

The first phase is executed by lines 1 through 7 to decompose (i, j) into a sequence C of $O(\log n)$ canonical index pairs (g, h) in a straightforward way. Obviously, this phase runs in $O(\log n)$ time.

The second phase determines indices $\kappa'(i, g, h)$ for all canonical index pairs (g, h) in C to determine $\kappa'(i, j)$ by executing lines 8 through 19. For each such (g, h) , lines 10 through 16 determine $\kappa'(i, g, h)$ based on Lemma 2 without having $K'(g, h)$ explicitly. The binary search in Lemma 2 is done by tracing the path from (g, h) to $(\kappa'(i, g, h), \kappa'(i, g, h))$. In this tracing process, two indices k_{\leftarrow} and k_{\rightarrow} are maintained so that whenever an internal vertex (e, f) is visited, for any index k with $e \leq k \leq f$, k is an element of $K'(g, h)$ if and only if

■ **Algorithm 2** A pseudo-code of Algorithm `determineKappa(i, j)`.

```

1 C ← an empty sequence;
2  $\tilde{j} \leftarrow 2^{\lceil \log_2 n \rceil}$ ;  $\tilde{i} \leftarrow 2^{\lceil \log_2 n \rceil} + 1$ ;
3 foreach  $l$  from  $\lceil \log_2 n \rceil$  to 1 in descending order do
4   if  $i \leq \tilde{j} - 2^l + 1$  and  $\tilde{j} \leq j$  then append  $(\tilde{j} - 2^l + 1, \tilde{j})$  to C;
5   if  $i \leq \tilde{j} - 2^l + 1$  then  $\tilde{j} \leftarrow \tilde{j} - 2^l$ ;
6   if  $i \leq \tilde{i}$  and  $\tilde{i} + 2^l - 1 \leq j$  then append  $(\tilde{i}, \tilde{i} + 2^l - 1)$  to C;
7   if  $\tilde{i} + 2^l - 1 \leq j$  then  $\tilde{i} \leftarrow \tilde{i} + 2^l$ ;
8  $\delta \leftarrow \infty$ ;
9 foreach  $(g, h)$  in C do
10   $l \leftarrow \log_2(h - g + 1)$ ;  $m \leftarrow h/2^l$ ;  $k_{\leftarrow} \leftarrow g$ ;  $k_{\rightarrow} \leftarrow h$ ;
11  while  $l > 0$  do
12     $k_{\leftarrow} \leftarrow K'[l - 1][2m - 1]$ ;  $k_{\rightarrow} \leftarrow K'[l - 1][2m]$ ;
13    if  $k_{\leftarrow} < k_{\rightarrow}$  or  $(k_{\leftarrow} \leq k_{\leftarrow}$  and  $\delta(i, k_{\leftarrow}) \leq \delta(i, k_{\rightarrow}))$  then
14       $k_{\leftarrow} \leftarrow \min(k_{\leftarrow}, k_{\rightarrow})$ ;  $l \leftarrow l - 1$ ;  $m \leftarrow 2m - 1$ 
15    else
16       $k_{\rightarrow} \leftarrow \max(k_{\leftarrow}, k_{\rightarrow})$ ;  $l \leftarrow l - 1$ ;  $m \leftarrow 2m$ 
17  if  $\delta(i, m) < \delta$  then
18     $\kappa' \leftarrow m$ ;  $\delta \leftarrow \delta(i, m)$ 
19 output  $\kappa'$  as  $\kappa'(i, j)$ 

```

$k_{\leftarrow} \leq k \leq k_{\rightarrow}$ and k is an element of $K'(e, f)$. Obviously, we can maintain k_{\leftarrow} and k_{\rightarrow} by initializing k_{\leftarrow} and k_{\rightarrow} to g and h , respectively, and updating k_{\leftarrow} (resp. k_{\rightarrow}) to the minimum (resp. maximum) of k_{\leftarrow} (resp. k_{\rightarrow}) and $k(e, f^*)$ (resp. $k(e^*, f)$), if (e, f^*) (resp. (e^*, f)) is chosen as the next vertex to visit after (e, f) , where $f^* = (e + f - 1)/2$ and $e^* = (e + f + 1)/2$. To guarantee that $e \leq \kappa'(i, g, h) \leq f$ for any vertex (e, f) visited in the trace, the next vertex to visit after (e, f) is chosen as follows. If $k_{\leftarrow} < k(e^*, f)$ (resp. $k(e, f^*) < k_{\rightarrow}$), then (e, f^*) (resp. (e^*, f)) is chosen, because $K'(e^*, f)$ (resp. $K'(e, f^*)$) shares no element with $K'(g, h)$. On the other hand, if both $k_{\leftarrow} \leq k(e, f^*)$ and $k(e^*, f) \leq k_{\rightarrow}$, then $k(e, f^*)$ and $k(e^*, f)$ are consecutive elements in $K'(g, h)$. Therefore, in such cases, it follows from Lemma 2 that we can choose (e, f^*) , if $\delta(i, k(e, f^*)) \leq \delta(i, k(e^*, f))$, or (e^*, f) , otherwise. If we represent (e, f) by indices $l = \log_2(f - e + 1)$ and $m = f/2^l$, then (e, f^*) (resp. (e^*, f)) is represented by $l - 1$ and $2m - 1$ (resp. $2m$), implying that $k(e, f^*) = K'[l - 1][2m - 1]$ (resp. $k(e^*, f) = K'[l - 1][2m]$). Adopting this representation, lines 10 through 16 trace the path from (g, h) to $(\kappa'(i, g, h), \kappa'(i, g, h))$ in the manner described above, and hence the resulting m represents $\kappa'(i, g, h)$. After this trace, κ' and δ are updated by lines 17 and 18 so that δ represents the maximum value of $\delta(i, \kappa'(i, g, h))$ for $\kappa'(i, g, h)$ obtained so far and κ' represents the first $\kappa'(i, g, h)$ found that satisfies $\delta(i, \kappa'(i, g, h)) = \delta$. Thus, line 19 outputs $\kappa'(i, j)$ correctly. The execution time of this phase is $O(\log^2 n)$ because the number of elements (g, h) in C is $O(\log n)$ and the number of vertices in the path from each such (g, h) to $(\kappa'(i, g, h), \kappa'(i, g, h))$ is $O(\log n)$. ◀

We define K'' as the forest K' for the reversed X_0 , which can be constructed in $O(n)$ time due to Lemma 3. Since $\kappa'(i, j)$ for the reversed X_0 represents $\kappa''((n + 1) - j, (n + 1) - i)$ for the original X_0 by symmetry, Lemma 4 implies that K'' can support $O(\log^2 n)$ -time queries of $\kappa''(i, j)$ for any index pair (i, j) with $1 \leq i \leq j \leq n$. Thus, the following theorem immediately follows from Lemmas 3 and 4.

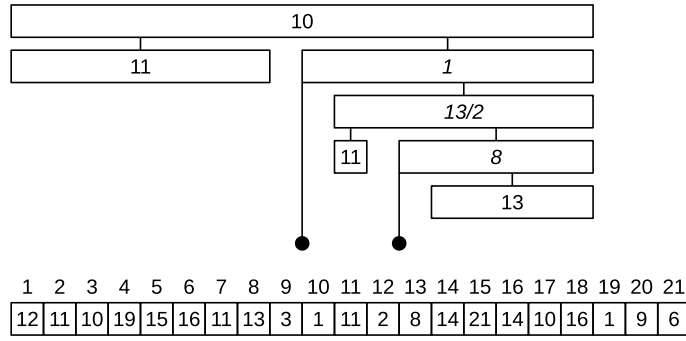


Figure 5 Tree $\tau(1, 18)$ for the same X_0 as Figure 2 shown at the bottom, where each vertex (g, h) with $g \leq h$ is indicated by a rectangle lying between positions g and h with $\alpha(g, h)$ as the label and each vertex (g, h) with $g = h + 1$ is represented as a bullet between positions h and g .

► **Theorem 5.** Forests K' and K'' can be constructed in $O(n)$ time and support $O(\log^2 n)$ -time queries of $\alpha(i, j)$ and $\kappa(i, j)$ for any index pair (i, j) with $1 \leq i \leq j \leq n$.

3.2 Offset-MSS data structure supporting $O(\log n)$ -time queries

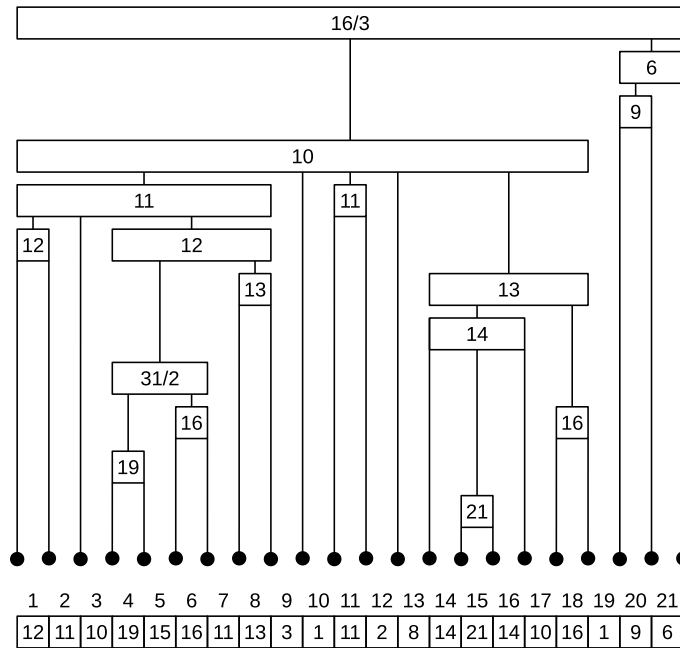
To design Algorithm `findOMSS`, we consider a tree T from which a vertex (i, j) can be chosen as an MSS of X_a for any real number a . After defining T based on Lemma 1, we analyze how the vertices of T used to construct $OMSS_{X_0}$ can be chosen.

Lemma 1 gives us the MSS of X_a specifically if $a < \alpha(1, n)$, but only inductive candidates otherwise. We define T as a tree that presents explicit rather than inductive candidates, even if $a \geq \alpha(1, n)$. More precisely, T is defined as a tree such that all vertices (g, h) with $\alpha(i, j) \leq a < \alpha(g, h)$ constitute the set of candidates, where (i, j) is the parent of (g, h) . Our idea to realize such a tree T is to divide the problem of finding an MSS of $X_a(i, j)$ for any real number a with $a \geq \alpha(i, j)$ into the problems of finding an MSS of $X_a(g, h)$ with $\alpha(g, h) > \alpha(i, j)$ by applying Lemma 1 incrementally to define the set of children (g, h) of each internal vertex (i, j) . Thus, formally, the set of children of any internal vertex (i, j) in T is defined as the set of leaves of the tree $\tau(i, j)$ introduced below. For any index pair (i, j) with $1 \leq i \leq j \leq n$, let $\tau(i, j)$ denote the tree such that (i, j) is the root, any vertex (g, h) with $g \leq h$ such that $X_{\alpha(i, j)}(g, h)$ is not pref/suff-positive (i.e., $\alpha(g, h) \leq \alpha(i, j)$) has two children $(g, \kappa(g, h) - 1)$ and $(\kappa(g, h) + 1, h)$, and any other vertex is a leaf (see Figure 5). Although the topology of $\tau(i, j)$ is not uniquely defined in general due to the ambiguity of $\kappa(g, h)$, it is not difficult to verify that the set of leaves is unique. Let T denote the tree such that the root is $(1, n)$, any vertex (i, j) with $i \leq j$ has all leaves of $\tau(i, j)$ as its children, and any other vertex is a leaf (see Figure 6). The following lemma claims that T has the property we intend.

► **Lemma 6.** For any real number a , if $a < \alpha(1, n)$, then $(1, n)$ is an MSS of X_a ; otherwise, any vertex (g, h) with $\alpha(i, j) \leq a < \alpha(g, h)$ in T that maximizes $S_a(g, h)$ is an MSS of X_a , where (i, j) is the parent of (g, h) .

Proof. If $a < \alpha(1, n)$, then the lemma immediately follows from Lemma 1. Suppose that $a \geq \alpha(1, n)$. Consider an arbitrary series \mathcal{C} of sets C of vertices in T such that

- the first set consists only of $(1, n)$,
- any set C containing at least one internal vertex (i, j) with $\alpha(i, j) \leq a$ as an element is followed by the set obtained from C by replacing arbitrary such element (i, j) with all children of $\tau(i, j)$, and
- any element (g, h) in the last set satisfies that $\alpha(g, h) > a$.



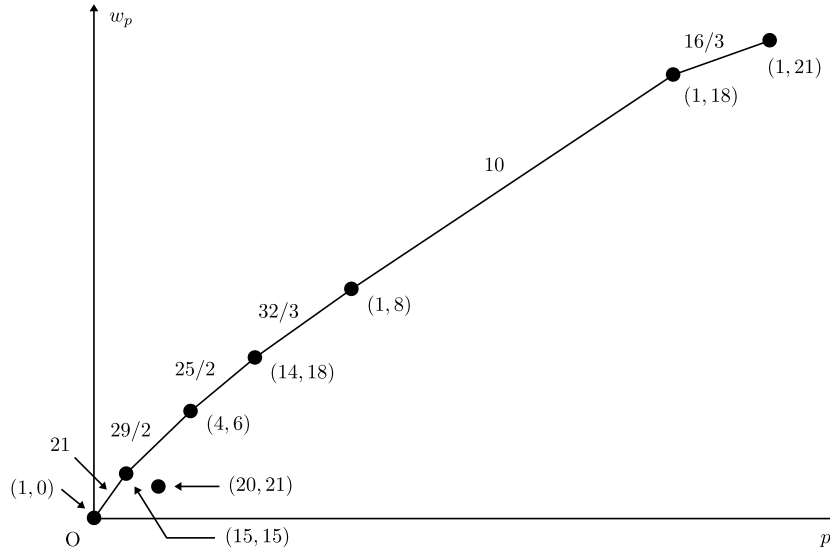
■ **Figure 6** Tree T for the same X_0 as Figure 5 shown at the bottom, drawn in the same manner as Figure 5.

Since $\alpha(g, h) > \alpha(i, j)$ for any leaf (g, h) of $\tau(i, j)$, the last set of \mathcal{C} exists and consists of all vertices (g, h) in T such that $\alpha(i, j) \leq a < \alpha(g, h)$, where (i, j) is the parent of (g, h) . It follows from Lemma 1 that any element (g, h) in the last set of \mathcal{C} is the only MSS of $X_a(g, h)$. Thus, if the last set in \mathcal{C} has an element (g, h) such that any MSS of $X_a(g, h)$ is an MSS of X_a , then the lemma holds. For any index pair (i, j) with $1 \leq i \leq j \leq n$, if $a \geq \alpha(i, j)$, then there exists a leaf (g, h) of $\tau(i, j)$ such that any MSS of $X_a(g, h)$ is an MSS of $X_a(i, j)$. This implies by induction that any set in \mathcal{C} has an element (g, h) such that any MSS of $X_a(g, h)$ is an MSS of X_a . ◀

Lemma 6 provides a set of vertices in T from which we can find an MSS of X_a . However, this candidate set varies with a . Furthermore, even if the same set is given for different real numbers a , the vertex (g, h) that is an MSS of X_a may differ from each other. For example, if we consider X_0 in Figure 6, then Lemma 6 claims that an MSS of $X_{12.4}$ and an MSS of $X_{12.6}$ can be found from the same candidates $(4, 6)$, $(8, 8)$, and $(14, 18)$ (ignoring empty candidates such as $(1, 0)$) and we can verify that $(14, 18)$ is the only MSS of $X_{12.4}$ while $(4, 6)$ is the only MSS of $X_{12.6}$. Instead of adopting the set of candidates suggested by Lemma 6 as is, we can consider a specific set of candidates that is not altered by a . Those candidates are introduced below. Let P denote the set of all lengths p with $0 \leq p \leq n$ such that T has at least one vertex (i, j) with $j - i + 1 = p$. For any length p in P , let w_p denote the maximum of $S_0(i, j)$ over all vertices (i, j) in T such that $j - i + 1 = p$ and let (i_p, j_p) denote an arbitrary such vertex (i, j) in T that achieves w_p . The following lemma claims that these vertices (i_p, j_p) can be thought of as the candidates.

► **Lemma 7.** For any real number a and any length p in P that maximizes $S_a(i_p, j_p)$, (i_p, j_p) is an MSS of X_a .

Proof. It immediately follows from Lemma 6 that any vertex (g, h) in T that maximizes $S_a(g, h)$ is an MSS of X_a . For any such vertex (g, h) , (i_p, j_p) is also an MSS of X_a , where $p = h - g + 1$, because $S_a(g, h) = S_0(g, h) - ap \leq w_p - ap = S_0(i_p, j_p) - ap = S_a(i_p, j_p)$. ◀



■ **Figure 7** Two-dimensional points (p, w_p) with (i_p, j_p) as the label for all lengths p in P used to determine $OMSS_{X_0}$ as the sequence of $(\infty, (1, 0))$, $(21, (15, 15))$, $(29/2, (4, 6))$, $(25/2, (14, 18))$, $(32/3, (1, 8))$, $(10, (1, 18))$, and $(16/3, (1, 21))$ in this order, where X_0 is the same as Figure 5 and each pair of adjacent vertices (q, w_q) and (p, w_p) of H is connected by a line with its slope $(w_p - w_q)/(p - q)$ as the label.

We are now ready to define $OMSS_{X_0}$ as the offset-MSS data structure we propose. Recall that $OMSS_{X_0}$ is a sequence of $O(n)$ pairs $(\theta, (i, j))$ in descending order of θ such that for any real number a , if $(\theta, (i, j))$ is the last element with $\theta > a$, then (i, j) is an MSS of X_a . As (i, j) of each such element $(\theta, (i, j))$, we adopt (i_p, j_p) for an appropriate length p in P . To find such lengths p , we treat (p, w_p) for any length p in P as a two-dimensional point and consider the upper convex hull H for all the points (p, w_p) (see Figure 7). Note that $(0, 0)$ is a vertex of H . We define $OMSS_{X_0}$ as consisting of pairs $(\theta_p, (i_p, j_p))$ for all vertices (p, w_p) of H . As threshold θ_p , we adopt ∞ , if $p = 0$, or $(w_p - w_q)/(p - q)$, otherwise, where (q, w_q) is the vertex of H that is adjacent to (p, w_p) such that $q < p$. The following theorem shows the correctness of $OMSS_{X_0}$ as an offset-MSS data structure.

► **Theorem 8.** *For any real number a , (i_p, j_p) is an MSS of X_a , where $(\theta_p, (i_p, j_p))$ is the last element in $OMSS_{X_0}$ such that $\theta_p > a$.*

Proof. Let q be arbitrary length in P such that $q < p$. Since (p, w_p) is a vertex of H , the slope $(w_p - w_q)/(p - q)$ of the straight line passing through (q, w_q) and (p, w_p) is greater than a . This implies that $S_a(i_q, j_q) = w_q - aq < w_p - ap = S_a(i_p, j_p)$. Analogously, for any length q in P such that $q > p$, $S_a(i_q, j_q) \leq S_a(i_p, j_p)$ because $(w_q - w_p)/(q - p) \leq a$. Thus, the theorem follows from Lemma 7. ◀

Algorithm `findOMSS` can be designed according to definition of $OMSS_{X_0}$. Algorithm 3 presents a pseudo-code of the algorithm, excluding the data structure proposed in Section 3.1 to support $O(\log^2 n)$ -time queries of $\alpha(i, j)$ and $\kappa(i, j)$. The algorithm consists of the following two phases.

The first phase. Lines 1 through 13 determine pairs (i_p, j_p) for all lengths p in P by enumerating all internal vertices (i, j) in T . During the enumeration, element $W[p]$ of array W for any length p with $1 \leq p \leq n$ is used to store the maximum of $S_0(i, j)$ over all vertices

■ **Algorithm 3** A pseudo-code of Algorithm findOMSS.

```

1 W, l, J  $\leftarrow$  arrays of  $n$  elements each initialized to 0;
2 T  $\leftarrow$  a stack containing a single element  $(1, n)$ ;
3 while T is non-empty do
4   pop  $(i, j)$  from T;
5   if  $W[p] < w$ , where  $p = j - i + 1$  and  $w = S_0(i, j)$ , then
6      $W[p] \leftarrow w$ ;  $l[p] \leftarrow i$ ;  $J[p] \leftarrow j$ 
7   tau  $\leftarrow$  a stack containing a single element  $(i, j)$ ;
8   while tau is non-empty do
9     pop  $(g, h)$  from tau;
10    if  $\alpha(g, h) \leq \alpha(i, j)$  then
11      push  $(g, \kappa(g, h) - 1)$  to tau; push  $(\kappa(g, h) + 1, h)$  to tau
12    else if  $g \leq h$  then
13      push  $(g, h)$  to T
14 H  $\leftarrow$  a sequence consisting of a single element  $(0, 0)$ ;
15 foreach  $p$  from 1 to  $n$  do
16   if  $W[p] > 0$  then
17     while H has more than one element and  $(W[p] - w_q)/(p - q) \geq (W[p] - w_r)/(p - r)$ ,
18       where  $(q, w_q)$  and  $(r, w_r)$  are the last and second last elements in H, respectively, do
19       delete the last element  $(q, w_q)$  from H
20   append  $(p, W[p])$  to H
21 OMSS  $\leftarrow$  a sequence consisting of a single element  $(\infty, (1, 0))$ ;
22 foreach  $(p, w_p)$  in H from the second element to the last in this order do
23   append  $((w_p - w_q)/(p - q), (l[p], J[p]))$  to OMSS, where  $(q, w_q)$  is the element
   immediately followed by  $(p, w_p)$  in H
23 output OMSS as  $OMSS_{X_0}$ 

```

(i, j) with $j - i + 1 = p$ in T enumerated up to the present time, if any, or 0, otherwise. In addition, elements $l[p]$ of array l and $J[p]$ of array J are used to indicate the first vertex (i, j) achieving $W[p]$. Therefore, after executing this phase, for any length p with $1 \leq p \leq n$, if p is an element of P , then $W[p]$, $l[p]$, and $J[p]$ represent w_p , i_p , and j_p , respectively; otherwise $W[p] = 0$. To enumerate all internal vertices in T , two stacks T and τ are used. Since there is no need to maintain the topology of T , T is used to store all internal vertices of T already found but whose children are not yet determined. After initializing T to a stack containing $(1, n)$ as the only element (by line 2), each such internal vertex (i, j) of T is popped from T (by line 4), treated as a new vertex found to update W , l , and J (by lines 5 and 6), and decomposed into its children (by lines 7 through 13). To decompose (i, j) into its children, τ is used to store all vertices of $\tau(i, j)$ that have found but not yet been determined to be leaves or not. After initializing τ to a stack containing (i, j) as the only element (by line 7), each element (g, h) is popped from τ (by line 9) and if (g, h) is not a leaf of $\tau(i, j)$, then its children are pushed to τ (by lines 10 and 11); otherwise, if (g, h) is an internal node of T , then (g, h) is pushed to T (by lines 12 and 13). For any internal vertex (i, j) of T and any internal vertex (g, h) of $\tau(i, j)$, there exists a distinct index k with $1 \leq k \leq n$ that corresponds to (g, h) . Thus, it follows from Lemma 4 that this phase is executed in $O(n \log^2 n)$ time.

The second phase. Lines 14 through 23 determine H to construct $OMSS_{X_0}$. For any length p with $1 \leq p \leq n$, let H_p be the upper convex hull for points (q, w_p) for all lengths q in P such that $q \leq p$, so that $H_n = H$. The sequence of all vertices (q, w) of H_p in ascending order

of q for each length p from 0 to n is inductively constructed as sequence H . After initializing H to a sequence consisting of $(0, w_0)$ as the only element (by line 14), for any length p in P other than 0 in ascending order, each element (q, w_q) in H that is not a vertex of H_p is removed one by one in descending order of q (by lines 17 and 18) and (p, w_p) is appended to H (by line 19). Since the sequence of all vertices (p, w) of H is eventually obtained as H , $OMSS_{X_0}$ is constructed as sequence $OMSS$ in a straightforward manner (by lines 20 through 22). To update H from H_0 to H_n , $(p, W[p])$ for any index p in P is appended to H exactly once and any such element is deleted from H at most once. Therefore, this phase is executed in $O(n)$ time.

Due to the above, together with Lemma 3, we immediately have the following theorem.

► **Theorem 9.** *Algorithm findOMSS, including the data structure supporting $O(\log^2 n)$ -time queries of $\alpha(i, j)$ and $\kappa(i, j)$ proposed in Section 3.1, outputs $OMSS_{X_0}$ as an $O(n)$ -space offset-MSS data structure for X_0 supporting $O(\log n)$ -time queries in $O(n \log^2 n)$ time and $O(n)$ space.*

4 Conclusive remarks

The present article considered the offset maximum-sum segment problem, a variant of the maximum-sum segment problem for a sequence X_0 of n real numbers, which asks an arbitrary contiguous subsequence of X_a that maximizes the sum of its elements for any given real number a , where X_a is the sequence obtained by subtracting a from each element in X_0 . An $O(n \log^2 n)$ -time, $O(n)$ -space algorithm that outputs a data structure supporting $O(\log n)$ -time queries of the solution of the offset maximum-sum segment problem was proposed. Further improvements in query time would be unlikely. This is because the data structure output by the proposed algorithm partitions the entire set of real numbers into $O(n)$ intervals, and the offset maximum-sum problem has a distinct solution in common for all real numbers a in each interval. It remains to be clarified whether or not the upper bound on the time complexity of finding such a data structure can be improved from $O(n \log^2 n)$.

References

- 1 Fredrik Bengtsson and Jingsen Chen. *Computing maximum-scoring segments optimally*. Luleå tekniska universitet, 2007.
- 2 Jon Bentley. Programming pearls: algorithm design techniques. *Communications of the ACM*, 27(9):865–871, 1984.
- 3 Kuan-Yu Chen and Kun-Mao Chao. On the range maximum-sum segment query problem. *Discrete Applied Mathematics*, 155(16):2043–2052, 2007. doi:10.1016/j.dam.2007.05.018.
- 4 Chih-Huai Cheng, Hsiao-Fei Liu, and Kun-Mao Chao. Optimal algorithms for the average-constrained maximum-sum segment problem. *Information Processing Letters*, 109(3):171–174, 2009. doi:10.1016/j.ipl.2008.09.024.
- 5 Kai-Min Chung and Hsueh-I Lu. An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, 34(2):373–387, 2005. doi:10.1137/S0097539704440430.
- 6 Margaret Dayhoff, R Schwartz, and B Orcutt. A model of evolutionary change in proteins. *Atlas of protein sequence and structure*, 5:345–352, 1978.
- 7 Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama. Data mining with optimized two-dimensional association rules. *ACM Transactions on Database Systems (TODS)*, 26(2):179–213, 2001. doi:10.1145/383891.383893.

- 8 Michael H Goldwasser, Ming-Yang Kao, and Hsueh-I Lu. Fast algorithms for finding maximum-density segments of a sequence with applications to bioinformatics. In *Algorithms in Bioinformatics: Second International Workshop, WABI 2002 Rome, Italy, September 17–21, 2002 Proceedings 2*, pages 157–171. Springer, 2002. doi:10.1007/3-540-45784-4_12.
- 9 Steven Henikoff and Jorja G Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992. doi:10.1073/pnas.89.22.10915.
- 10 Sung Kwon Kim. Linear-time algorithm for finding a maximum-density segment of a sequence. *Information Processing Letters*, 86(6):339–342, 2003. doi:10.1016/S0020-0190(03)00225-4.
- 11 Yaw-Ling Lin, Tao Jiang, and Kun-Mao Chao. Efficient algorithms for locating the length-constrained heaviest segments with applications to biomolecular sequence analysis. *Journal of Computer and System Sciences*, 65(3):570–586, 2002. doi:10.1016/S0022-0000(02)00010-7.
- 12 Kalyan Perumalla and Narsingh Deo. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, 5(03):367–373, 1995. doi:10.1142/S0129626495000345.
- 13 Walter L Ruzzo and Martin Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *ISMB*, volume 99, pages 234–241, 1999. URL: <http://www.aaii.org/Library/ISMB/1999/ismb99-027.php>.
- 14 Yoshifumi Sakai. A maximal local maximum-sum segment data structure. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 101(9):1541–1542, 2018. doi:10.1587/transfun.E101.A.1541.
- 15 Nikola Stojanovic, Liliana Florea, Cathy Riemer, Deborah Gumucio, Jerry Slightom, Morris Goodman, Webb Miller, and Ross Hardison. Comparison of five methods for finding conserved sequences in multiple alignments of gene regulatory regions. *Nucleic Acids Research*, 27(19):3899–3910, 1999. doi:10.1093/nar/27.19.3899.
- 16 Lusheng Wang and Ying Xu. Segid: Identifying interesting segments in (multiple) sequence alignments. *Bioinformatics*, 19(2):297–298, 2003. doi:10.1093/bioinformatics/19.2.297.
- 17 Hung-I Yu, Tien-Ching Lin, and DT Lee. Finding maximum sum segments in sequences with uncertainty. *Theoretical Computer Science*, 850:221–235, 2021. doi:10.1016/j.tcs.2020.11.005.