Linear Time Computation of the Maximal (Circular) Sums of Multiple Independent Insertions of Numbers into a Sequence*

Pablo M. S. Farias† Ricardo C. Corrêa

ParGO Research Group‡
Universidade Federal do Ceará, Campus do Pici, Bloco 910, 60440-554 Fortaleza, CE, Brazil
{pmsf,correa}@lia.ufc.br

Abstract

The maximal sum of a sequence $A$ of $n$ real numbers is the greatest sum of all elements of any strictly contiguous and possibly empty subsequence of $A$, and it can be computed in $O(n)$ time by means of Kadane’s algorithm. Letting $A^{(x\rightarrow p)}$ denote the sequence which results from inserting a real number $x$ between elements $A[p+1]$ and $A[p]$, we show how the maximal sum of $A^{(x\rightarrow p)}$ can be computed in $O(1)$ worst-case time for any given $x$ and $p$, provided that an $O(n)$ time preprocessing step has already been executed on $A$. In particular, this implies that, given $m$ pairs $(x_0,p_0),\ldots,(x_{m-1},p_{m-1})$, we can compute the maximal sums of sequences $A^{(x_0\rightarrow p_0)},\ldots,A^{(x_{m-1}\rightarrow p_{m-1})}$ in $O(n+m)$ time, which matches the lower bound imposed by the problem input size, and also improves on the straightforward strategy of applying Kadane’s algorithm to each sequence $A^{(x_i\rightarrow p_i)}$, which takes a total of $\Theta(n\cdot m)$ time. Our main contribution, however, is to obtain the same time bound for the more complicated problem of computing the greatest sum of all elements of any strictly or circularly contiguous and possibly empty subsequence of $A^{(x\rightarrow p)}$. Our algorithms are easy to implement in practice, and they were motivated by and find application in a buffer minimization problem on wireless mesh networks.

Keywords: Maximal sum subsequence, Multiple insertions into a sequence, Circular subsequence, FIFO order, Priority queue.

1 Introduction

Our aim in this paper is to provide efficient algorithms to answer certain insertion-related queries on a sequence of numbers. For a given sequence $A$ of $n$ real numbers, a query takes as arguments a real number $x$ and an index $p \in \{0,\ldots,n\}$, and returns the “cost” of sequence $A^{(x\rightarrow p)}$, the latter being the sequence which results from inserting $x$ between elements $A[p-1]$ and $A[p]$. By the “cost” of a sequence $B$ of real numbers we mean the greatest sum of all elements of any contiguous, possibly empty subsequence $S$ of $B$. Our focus in this paper is on independent queries, that is, given a fixed sequence $A$, we want to answer a number of (a priori unrelated) queries on the same sequence $A$.

1.1 Definitions and Results

We denote an arbitrary sequence of $n$ elements by $A = \langle A[0],\ldots,A[n-1] \rangle$ and its size by $|A| = n$. Noncircular subsequences are denoted as follows: $A[i:j] = \langle A[i], A[i+1],\ldots,A[j] \rangle$ if $0 \leq i \leq j < n$, otherwise $A[i:j] = \langle \rangle$ (empty sequence). The concatenation of sequences $A$ and $B$ of sizes $n$ and $m$ is denoted by $A + B = \langle A[0],\ldots,A[n-1],B[0],\ldots,B[m-1] \rangle$. The sequence which results from the insertion of an element $x$ into position $p \in \{0,\ldots,n\}$ of a sequence $A$ of size $n$ is denoted by $A^{(x\rightarrow p)}$.

*This work is partially supported by FUNCAP/INRIA (Ceará State, Brazil/France) and CNPq (Brazil) research projects.

†Partially supported by a CAPES doctoral scholarship (“Programa de Demanda Social”).

‡http://www.lia.ufc.br/~pargo
or simply by $A(p) = A[0 : p - 1] + \langle x \rangle + A[p : n - 1]$. If $A$ is a sequence of real numbers, then its sum is $\text{sum}(A) = \sum_{i=0}^{n-1} A[i]$, which equals zero if $A = \langle \rangle$. Moreover, the maximal subsequence sum or simply maximal sum of $A$, denoted by $\text{MS}(A)$, is the greatest sum of a noncircular subsequence of $A$. Note that, since $\langle \rangle$ is subsequence of any sequence, then $\text{MS}(A)$ is always nonnegative.

Let MAXIMAL SUMS OF INDEPENDENT INSERTIONS (MSII) be the problem of, given a sequence $A$ of $n$ real numbers and $m$ pairs $(x_0, p_0), \ldots, (x_{m-1}, p_{m-1})$, with $x_i \in \mathbb{R}$ and $p_i \in \{0, \ldots, n\}$, computing the maximal sums of sequences $A(x_0 \rightarrow p_0), \ldots, A(x_{m-1} \rightarrow p_{m-1})$. In this paper, we show that the MSII problem can be solved in $O(n + m)$ time. Note that the size of the input to the problem implies that it cannot be solved in an asymptotically smaller time: any output given by an algorithm which has not read all of $A[0], \ldots, A[n-1]$ and all of $x_0, \ldots, x_{m-1}$ can be refuted by an adversary who chooses sufficiently large values for the input numbers which have not been read. It follows that our solution to the MSII problem is time optimal.

We also extend the result above in order to take circular subsequences into account. More precisely, given a sequence $A$ of size $n$, we define the possibly circular subsequence of $A$ induced by indices $i$ and $j$ as $A[i : j] = A[i : n - 1] + A[0 : j]$, if $0 \leq j < i < n$, and as $A[i : j] = A[i : j]$, otherwise. Moreover, if $A$ is a sequence of real numbers, we define the maximal circular sum of $A$, denoted by $\text{MCS}(A)$, as the greatest sum of a possibly circular (and thus also possibly empty) subsequence of $A$. We then show in this paper that the “circular” variation of the MSII problem, in which one is requested to compute the maximal circular sum of each sequence $A(x_i \rightarrow p_i)$, can also be solved in $O(n + m)$ time, which again is a time optimal solution.

Our solution to the MSII problem is composed of four algorithms, two for the noncircular case and two for the circular case. In both cases, one of the two algorithms, which may be seen as performing a preprocessing step, takes as argument a sequence $A$ of $n$ real numbers and computes several arrays, each storing some kind of information about the sequence – in the noncircular case, for example, one such array is called “MS” and is defined by $\text{MS}[i] = \max\{\text{sum}(A[j : i]) : j \in \{0, \ldots, i\}\}$ for every $i \in \{0, \ldots, n - 1\}$. The other algorithm is then a query-answering one: it takes as arguments a real number $x$ and an index $p \in \{0, \ldots, n\}$ and, using the arrays produced by the first algorithm, computes $\text{MS}(A(x \rightarrow p))$, in the noncircular case, or $\text{MCS}(A(x \rightarrow p))$, in the circular case. In both cases, the first algorithm takes $O(n)$ time and the second one takes $O(1)$ time – here, and also anywhere else unless otherwise specified, we mean worst-case time. Our $O(n + m)$ time solution to the MSII problem now follows immediately: given an input $(A, (x_0, p_0), \ldots, (x_{m-1}, p_{m-1}))$, we first perform an $O(n)$ time preprocessing step on $A$ and then use the query-answering algorithm to compute the maximal (circular) sum of $A(x_i \rightarrow p_i)$ in $O(1)$ time for each $i \in \{0, \ldots, m - 1\}$.

1.2 Motivation and Applications

The algorithms proposed in this paper were motivated by a buffer minimization problem in wireless mesh networks, as follows. In a radio network, interference between nearby transmissions prevents simultaneous communication between pairs of nodes which are sufficiently close to each other. One way to circumvent this problem is to use a time division multiple access (TDMA) communication protocol. In such a protocol, the communication in the network proceeds by the successive repetition of a sequence of transmission rounds, in each of which only noninterfering transmissions are allowed to take place. Such protocols can also be used in the particular case of a wireless mesh network, where each node not only communicates data relevant to itself, but also forwards packets sent by other nodes, thus enabling communication between distant parts of the network [1, 2]. In this case, each node stores in a buffer the packets that it must still forward, and optimizing buffer usage in such networks is a current research topic [3, 4].

In order to analyze the use of buffer in a given set of $m$ nodes of a network, we represent a sequence of $n$ transmission rounds by an $m \times n$ matrix $A$, defined as follows: $A[i, j] = +1$ if on round $j$ node $i$ receives a packet that must be forwarded later (this node therefore needs one additional unit of memory after that round), $A[i, j] = -1$ if on round $j$ node $i$ forwards a packet (and thus needs one less unit of memory after that round), and $A[i, j] = 0$ otherwise; note that the sum of each row of $A$ must be zero, since nodes can forward neither less nor more packets than they receive for this. Now, since the same sequence of rounds is successively repeated in the network, then the
variation of buffer usage for node \( i \) on the \( k \)-th “global” transmission round, with \( k \in \mathbb{N} \), is given by \( f_i(k) = A[i, k \mod n] \). Moreover, the greatest number of packets that node \( i \) will ever need to store simultaneously is the greatest sum of a subsequence of the infinite sequence \( \langle f_i(0), f_i(1), f_i(2), \ldots \rangle \), which, by the repetitive nature of this sequence and since the sum of row \( i \) of \( A \) is zero, equals the maximal circular sum of row \( i \) of \( A \). The use of buffer for the whole set of \( m \) nodes is then given by 
\[
\text{cost}(A) = \sum_{i=0}^{m-1} \text{MCS}(\langle A[i,0], \ldots, A[i,n-1] \rangle).
\]
This leads us to the following problem: given a matrix \( A \) representing an arbitrary ordering of the elements of a multiset of transmission rounds – such a multiset may for example be induced by a solution to the round weighting problem defined in [2] –, find a permutation \( A' \) of the columns of \( A \) which minimizes \( \text{cost}(A') \).

The problem defined above is NP-hard; actually, if we allow the input matrix to have integers not in \( \{-1, 0, +1\} \), then the problem is strongly NP-hard even if \( A \) is restricted to have only one row and if circular subsequences are not considered in the cost function [5]. However, the algorithms introduced in the present paper provide a valuable tool for the development of heuristics to the problem. To see why this is the case, consider the operation of moving column \( k \) of matrix \( A \) to some other position in the matrix, in such a way that the cost of the resulting matrix is minimized. Such an operation can clearly be used in an heuristic to the problem, for example in a local search algorithm in which two elements of the search space are said to be neighbours if they can be obtained one from the other by the operation in question. Now, what is the time complexity of this operation? Let then \( B \) be the \( m \times n \) matrix obtained by removing column \( k \) of \( A \), and let \( C \) be the \( m \times (n+1) \) matrix such that \( C[i,j] = \text{MCS}(B[i,A[k-j]]) \), where \( B_i \) denotes row \( i \) of \( B \). Clearly, inserting column \( k \) of \( A \) between columns \( j-1 \) and \( j \) of \( B \) produces a matrix whose cost is \( \sum_{i=0}^{m-1} C[i,j] \). Moreover, by using our preprocessing and query-answering algorithms for the circular case, we can compute matrix \( C \) in \( O(m \cdot n) \) time. It follows that the operation in question can be implemented in linear time.

The column permutation problem defined above, as well as solutions to it based on the algorithms proposed here, will be the topic of an upcoming paper. In the current paper, the focus is on our preprocessing and query-answering algorithms, which are a contribution in their own. In this regard, observe that the concept of maximal sum subsequence and its generalization for two dimensions are currently known to have several other applications in practice, for example in Pattern Recognition [6, 7], Data Mining [8], Computational Biology [9, 10], Health and Environmental Science [11] and Strategic Planning [12]; consequently, it is plausible that other applications of our algorithms be found in the future.

### 1.3 Related Work

The basic problem of finding a maximal sum subsequence of a sequence \( A \) of \( n \) numbers was given an optimal and very simple solution by Joseph B. Kadane around 1977; the algorithm, which takes only \( O(n) \) time and \( O(1) \) space, was discussed and popularized by Gries [13] and Bentley [6]. This one-dimensional problem can be generalized for any number \( d \) of dimensions. The two-dimensional case, which was actually the originally posed problem [6], consists in finding a maximal sum submatrix of a given \( m \times n \) matrix of numbers, and it can be solved in \( O(m^2 \cdot n) \) time, with \( m \leq n \) [14]; asymptotically slightly faster algorithms do exist but are reported not to perform well in practice except for very large inputs [11, 7].

Another direction of generalization of the original problem which has been explored is that of finding multiple maximal sum subsequences instead of just one. In the ALL MAXIMAL SCORING SUBSEQUENCES problem, one must find a set of all successive and nonintersecting maximal sum subsequences of a given sequence \( A \) of \( n \) numbers; this problem can be solved in \( O(n) \) sequential time [9], \( O(\log n) \) parallel time in the EREW PRAM model [15] and \( O(|A|/p) \) parallel time with \( p \) processors in the BSP/CGM model [16]. A different problem, concerning the maximization of the \( \text{sum} \) of any set of \( k \) nonintersecting subsequences, is considered in [10]. In the \( k \) MAXIMAL SUMS problem, on the other hand, one must find a list of the \( k \) possibly intersecting maximal sum subsequences of a given sequence of \( n \) numbers, which can be done in optimal \( O(n + k) \) time and \( O(k) \) space [17]. In the related SUM SELECTION problem, one must find the \( k \)-th largest sum of a subsequence of a given sequence \( A \) of \( n \) numbers, which can be done in optimal \( O(n \cdot \max\{1, \log(k/n)\}) \) time [18]. Some of these problems have also been considered in the length constrained setting, where only subsequences
of size at least \( l \) and at most \( u \) of the input sequence are considered [18, 19].

To the best of our knowledge, the column permutation problem defined in the previous subsection has not yet been considered in the literature. The closest related and already studied problem that we know of is the following variation of it for only one row: given a sequence \( A \) of \( n \) real numbers, find a permutation \( A' \) of \( A \) which minimizes \( \mathcal{M}S(A') \). This problem was found to be solvable in \( O(\log n) \) time in the particular case where \( A \) has only two distinct numbers [20]; the same paper also mentions that the case where \( A \) may have arbitrary numbers can be shown to be strongly NP-hard by reduction from the 3-PARTITION problem. Such a reduction has actually been presented recently, together with an \( O(n \log n) \) algorithm which has an approximation factor of 2 for the case of arbitrary input numbers and 3/2 for the case where the input numbers are subject to certain restrictions [5].

Problems about insertion-related operations in a sequence of numbers in connection with the concept of maximal subsequence sum seem to have been considered only in recent papers by the present authors, in which restricted versions of the noncircular case of the MSII problem were dealt with. Precisely, in a first paper we considered the problem of, given a sequence \( A \) of \( n \) real numbers and a real number \( x \), finding an index \( p \in \{0, \ldots, n\} \) which minimizes \( \mathcal{M}S(A^{(x\rightarrow p)}) \), and we showed that this can be done by means of a simple linear time algorithm [5]. Later we generalized this result by considering the problem of, given sequences \( A \) and \( X \) of \( n \) and \( n + 1 \) real numbers respectively, computing \( \mathcal{M}S(A^{(X[p]\rightarrow p)}) \) for all \( p \in \{0, \ldots, n\} \), and for this problem we also gave an \( O(n) \) time algorithm [21].

1.4 Contributions of this Paper

Our first contribution in this paper is our solution to the noncircular version of the MSII problem. This solution generalizes the algorithm given in [21] in order to handle any number \( m \) of queries in a sequence of \( n \) numbers, each query concerning an arbitrary insertion position \( p \) in the sequence, while our previous algorithm can only handle the fixed number of \( n \) queries in the sequence, one for each insertion position \( p \in \{0, \ldots, n\} \). Although much of the essence of the new algorithm is shared by the old one, we add and work out the important observation that our previous algorithm can be split into two steps, namely an \( O(n) \) time preprocessing step and a constant worst-case time query-answering step which works for any \( p \), which is what yields the solution to the more general MSII problem. Note also that, while our solution to the MSII problem runs in optimal \( O(n + m) \) time, the straightforward strategy of solving the problem by running Kadane’s algorithm \( m \) times takes \( \Theta(n \cdot m) \) time.

Our second contribution in this paper is our solution to the circular case of the MSII problem, which consists in extending the result above in order to take circular subsequences into account. This extension turned out not to be a straightforward one, having demanded two important additions: in the first place, different kinds of algorithms were introduced in the preprocessing step, in order to produce the data needed by the query-answering algorithm for the circular case; in the second place, the structural information provided by Kadane’s algorithm about the input sequence, which we call the interval partition of the sequence [5] and which is fundamental for our algorithm in the noncircular case, was found not to be enough in the circular case, and a nontrivial generalization of it was developed. For these reasons, the preprocessing and query-answering algorithms which we give for the circular case are the main contribution of this paper. The computational implications of these algorithms are the same as those for the noncircular case: it is easy to modify Kadane’s algorithm so that it also takes circular subsequences into account, but using it to solve the circular case of the MSII problem leads to a \( \Theta(n \cdot m) \) time solution, while our algorithms solve the problem in optimal \( O(n + m) \) time; note, in particular, that this implies an improvement from \( \Theta(m \cdot n^2) \) to \( \Theta(m \cdot n) \) in the time complexity of the column moving operation defined in Subsection 1.2.

Our last contribution in this paper is actually a by-product of our preprocessing algorithm for the circular case: the kind of computation performed in this algorithm lead us to develop a new simple data structure, which we dubbed the max-queue. A max-queue is a data structure in which, like in any priority queue, each element has a key associated with it, but which, unlike common priority queues, is subject to the following constraints: (1) elements are removed in FIFO (“first in, first out”) order, not by the values of their keys; moreover, when an element is removed, its key and satellite data need not be returned; (2) it must be possible to peek, without removing, the element
with greatest key; in case two or more elements have the greatest key, this operation refers to the oldest (i.e. least recently inserted) such element; (3) the insertion operation takes as arguments not only a new key and its satellite data, but also some real number \( d \), which must be added to the keys of all elements currently stored in the data structure before the new key is inserted. Clearly, what makes the max-queue not trivial to implement is the extra argument \( d \) of its insertion operation, since otherwise it could immediately be implemented by means of a standard queue. Our implementation of the max-queue is such that the removal and peeking operations take constant worst-case time, and such that the insertion operation takes constant amortized time. This data structure was crucial in our preprocessing algorithm for the circular case, and it may happen to find other uses in the future.

### 1.5 Structure of the Paper

The remaining of this paper is structured as follows. Section 2 introduces our approach as well as some notation for both the noncircular and the circular case. The preprocessing and query-answering algorithms for the noncircular case are then presented in Sections 3 and 4, respectively. Section 5 presents our extension of the interval partition concept and other preliminaries for the circular case. Section 6 presents the max-data queue data structure. Sections 7 and 8 then present our preprocessing and query-answering algorithms for the circular case, and Section 9 presents our concluding remarks, closing the paper.

### 2 General Approach

We begin with a few general definitions. Given \( a, b \in \mathbb{N} \), we denote the range of natural numbers from \( a \) to \( b \) by \( [a .. b] = \{ i \in \mathbb{N} : a \leq i \leq b \} \). Moreover, for any sequence \( S \) of size \( m \geq 1 \) and \( i \in [0 .. m - 1] \), we say that \( S[0 : i] \) and \( S[i : m - 1] \) are respectively a prefix and a suffix of \( S \). A prefix or suffix of \( S \) is said to be proper if it is different from \( S \).

#### 2.1 Noncircular Case

Let \( A \) be a sequence of \( n \) real numbers, \( x \in \mathbb{R} \), \( p \in [0 .. n] \) and recall that \( A^{(p)} \) abbreviates \( A^{(x \rightarrow p)} \). In order to compute \( MS(A^{(p)}) \) fast, we need to find out quickly how the insertion of \( x \) relates to the sums of the subsequences of \( A \); thus, in a sense, we need to previously know the “structure” of \( A \). A simple and useful characterization of such a structure of \( A \) is implicit in Kadane’s algorithm. Indeed, defining the maximal sum at element \( A[i] \) as \( MS_A(i) = \max \{ \text{sum}(A[j : i]) : 0 \leq j \leq i \} \) for all \( i \in [0 .. n - 1] \), it immediately follows that \( MS(A) = \max \{ \text{sum}(\emptyset) \} \cup \{ MS_A(i) : 0 \leq i < n \} \). Kadane’s algorithm then computes \( MS(A) \) in a single left-to-right sweep of \( A \) by observing that \( MS_A(0) = A[0] \) and that \( MS_A(i + 1) = A[i + 1] + \max\{0, MS_A(i)\} \) for all \( i \). Note that, in this setting, an element \( A[i] \) such that \( MS_A(i) < 0 \) corresponds to a “discontinuity” in the computation of the maximal sums at the elements of \( A \), and that this induces a partition of the sequence into “intervals” where this computation is “continuous”. More precisely, let the interval partition of \( A[5] \) be its unique division into nonempty subsequences \( I_0, \ldots, I_{\ell - 1} \), with \( I_i = A[\alpha_i : \beta_i] \), such that \( A = I_0 + \ldots + I_{\ell - 1} \) and such that, for every interval \( I_i \) and index \( j \in [\alpha_i .. \beta_i] \), \( j \neq \beta_i \Rightarrow MS_A(j) \geq 0 \) and \( j = \beta_i \) and \( i < \ell - 1 \Rightarrow MS_A(j) < 0 \). Note that this definition implies that \( MS_A(j) = \text{sum}(A[\alpha_i : j]) \) and that \( (i \neq \ell - 1 \Rightarrow \text{sum}(A[j : \beta_i]) < 0 \) for every interval \( I_i \) and index \( j \in [\alpha_i .. \beta_i] \).

Fig. 1 depicts the interval partition of a particular sequence and an insertion into it; note the use of index \( k \): throughout the paper, \( I_k \) denotes the interval such that \( p \in [\alpha_k .. \beta_k] \), if \( p < n \), or such that \( k = \ell - 1 \), if \( p = n \). An immediate observation is then that the insertion of \( x \) never affects the maximal sum at any element \( A[i] \) such that \( i < p \). Clearly, thus, if we define \( PRVMS \) as the array such that \( PRVMS[i] = \max\{0, MS_A(j) : 0 \leq j < i\} \) for all \( i \in [0 .. n] \), then

\[
\max\{0\} \cup \{ MS_{A^{(p)}}(i) : 0 \leq i < p \} = PRVMS[p].
\]

Note that the interval partition of \( A \) and array \( PRVMS \) can be computed on the fly by Kadane’s algorithm, so we can compute them in our \( O(n) \) time preprocessing algorithm. In general, however, the effect of the insertion of \( x \) on the remaining elements of \( A \) actually depends on the sign of \( x \). Taking
example, this is the case for the first four elements of sequence

$A.$ Let the

index

part. As indicated above, the latter step only depends on a

fixed

number of data associated with

$x$ and computing the greatest maximal sum at an element of each

Thus, if we define

$MS$ as the array such that $SFMS[i] = \max\{MS_A(j) : i \leq j \leq \beta_{K[i]}\}$ for all

$i \in [0..n-1]$, that

$p < n$ implies

$$\max\{MS_A(i^o) : i \in [p..\beta_k]\} = x + SFMS[p].$$

Note that array $SFMS$ can easily be computed in a single right-to-left traversal of array $MS$ by taking the interval partition of $A$ into account.

Handling elements $A[\beta_k + 1], \ldots, A[n-1]$ when $x \geq 0$ is more complicated, since in general the maximal sums at these elements do not all vary by the same amount; the same is true of elements $A[p], \ldots, A[\beta_k]$ when $x < 0$. We defer these more elaborate analyses of the noncircular case to the next two sections. However, the considerations above already give an outline of our approach, which, as depicted in Fig. 1, consists in dividing sequence $A^{(p)}$ into four parts (whose elements behave similarly with regard to the insertion of $x$) and computing the greatest maximal sum at an element of each part. As indicated above, the latter step only depends on a fixed number of data associated with index $p$ ($PRVMS[p]$, $SFMS[p]$, etc), which is what yields a constant time computation of $MS(A^{(p)})$. Moreover, by applying Dynamic Programming to avoid repeating computations, our preprocessing algorithm is able to compute the necessary data for all insertion positions (i.e. a fixed number of $O(n)$-sized arrays) collectively in $O(n)$ time, yielding the desired time bounds.

### 2.2 Circular Case

Let the maximal circular sum at element $A[i]$ be defined as $MCS_A(i) = \max\{\sum(A[j : i]) : 0 \leq j < n\}$ for all $i \in [0..n-1]$. A first novelty in the circular case is then that, for any element $A[i]$ with $i \neq n-1$, it may happen that no index $j \in [0..i]$ be such that $\sum(A[j : i]) = MCS_A(i)$; as an example, this is the case for the first four elements of sequence $A'$ from Fig. 2. However, since this never happens if $MCS_A(n-1) < 0$, then, as long as there is some index $i$ such that $MCS_A(i) < 0$, this situation may be avoided by performing a suitable circular shift on $A$ during the preprocessing step and updating insertion position $p$ accordingly in the query-answering algorithm (clearly, this does not affect the answer returned by the latter algorithm, since circularly shifting a sequence does not change its maximal circular sum).

Let us take the case where there is some $i$ such that $MCS_A(i) < 0$ as our first example. As argued above, we may suppose that $A$ has already been properly shifted and thus that $MCS_A(n-1) < 0$. An important observation is then that, in this case, $MCS_A(i) = MS_A(i)$ for all $i \in [0..n-1]$, that is, the interval partition of $A$ is the same in the circular and noncircular cases. This is also true of sequence $A^{(p)}$ if $x < 0$, in which case we can compute $MCS(A^{(p)})$ exactly as we compute $MS(A^{(p)})$ in

![Figure 1: Insertion in the noncircular case, with $x = 12$ and $p = 8$. Elements have equal colors iff they belong to the same interval in $A$. Above each element is its maximal sum in the sequence; the greatest such values are written in bold for each sequence. Below $A^{(p)}$ is indicated its division into four parts which is used by the query-answering algorithm for the noncircular case.](image-url)
Figure 2: Insertion in the circular case, with $x = 12$ and $p = 10$. $A$ is the sequence which results from a circular shift of two positions to the right performed on $A'$. Elements have equal colors iff they belong to the same interval in $A$. Above each element is its maximal circular sum in the sequence; the greatest such values are written in bold for each sequence. Below $A^{(p)}$ is indicated its division into four parts which is used by the query-answering algorithm for the circular case.

the noncircular case. As exemplified in Fig. 2, however, this does not hold in general if $x \geq 0$, since the insertion of $x$ may also affect the maximal circular sums at elements $A[0], \ldots, A[p-1]$. Thus, when $x \geq 0$, we divide $A^{(p)}$ into four parts as in the noncircular case – but with a suitably modified division of the sequence – and then compute the greatest value of $\text{MCS}_{A^{(p)}}(i)$ for all $i$ in each part. We defer the details of this analysis to Section 8. With regard to the preprocessing step, however, note that, while computing $\text{MS}_{A}(0), \ldots, \text{MS}_{A}(n-1)$ in linear time is easy, computing $\text{MCS}_{A}(0), \ldots, \text{MCS}_{A}(n-1)$ within the same time bound is more complicated. The reason is that, although each term $\text{MCS}_{A}(i)$ takes all elements of $A$ into account, these elements are combined differently for each index $i$. The situation is similar with regard to other data (i.e. arrays) needed by the query-answering algorithm for the circular case, which demands a different Dynamic Programming strategy in the preprocessing algorithm. A simple yet general framework to support this strategy is provided by the max-queue data structure, which we define in Section 6 and use in Section 7.

It may also be the case that $A$ has no element $A[i]$ such that $\text{MCS}_{A}(i) < 0$. This case can be trivially handled if $A$ happens not to have negative elements. However, if $A$ has both positive and negative elements, then the problem is not trivial and, furthermore, the definition of interval partition we used before, in which an interval ends exactly when the maximal sum at one of its elements falls below zero, does not suffice anymore. We discuss this issue in detail in Section 5, where we present a different criterion for delimiting intervals; this criterion applies uniformly to both the present case and the previous one of negative maximal circular sums in $A$, and it also naturally extends the one used in the noncircular case. Even so, however, the different “structure” of sequence $A$ when $\text{MCS}_{A}(i) \geq 0$ for all $i$ demands a different analysis in the query-answering algorithm, particularly when $x < 0$; fortunately, though, this new structure is also quite regular, enabling a constant time computation of $\text{MCS}(A^{(p)})$ by means of data which can be computed in linear time during the preprocessing step, as desired.

As explained in the previous paragraphs, our computation of $\text{MCS}(A^{(p)})$ in the query-answering algorithm depends not only on the sign of $x$, but also on the type of sequence $A$. For convenience, we say that sequence $A$ is of type 1 if it has only nonnegative or only nonpositive elements; type 2 if it has both positive and negative elements and furthermore is such that $\text{MCS}_{A}(i) < 0$ for some $i$; finally, type 3 otherwise, that is, if it has both positive and negative numbers and is such that $\text{MCS}_{A}(i) \geq 0$ for all $i$.

We close this section with a remark. In face of the increased complexity of our algorithms for the circular case, one might wonder whether the simpler algorithms for the noncircular case cannot actually be used in the circular setting. One idea that often comes to the mind is to take both circular and noncircular subsequences of $A$ into account by considering the noncircular subsequences of sequence $A + A$. Note, however, that this demands some provision in order to avoid that subsequences of size
then to show that the remaining arrays of Table 1: Definitions of the arrays used in the noncircular case. Those indexed with \( x \) are not defined for every index. The following abbreviations are used: 

\[
\begin{align*}
K[i] &= \begin{cases} 
\ell - 1 & \text{if } i = n \\
\text{the } j \in [0 .. \ell - 1] \text{ such that } i \in [\alpha_j .. \beta_j] & \text{if } i < n
\end{cases} \\
PRVMS[i] &= \max\{0\} \cup \{\mathcal{M}_A(j) : 0 \leq j < i\}
\end{align*}
\]

Arrays with indices in the range \([0 .. n - 1]\)

\[
\begin{align*}
MS[i] &= \mathcal{M}_A(i) \\
SFMS[p] &= \max\{\mathcal{M}_A(i) : p \leq i \leq \beta_k\} \\
XI[p] &= \max\{i \in [p .. x^*] : \mathcal{M}_A(i) \geq \mathcal{M}_A(j) \text{ for all } j \in [p .. \beta_k]\} \\
X2[p] &= \text{as computed by Algorithm 1.} \\
BX1[p] &= \min\{\mathcal{M}_A(j) : p \leq j < x_1\} \\
BX2[p] &= \min\{\mathcal{M}_A(j) : p \leq j < x_2\}
\end{align*}
\]

Arrays with indices in the range \([0 .. \ell - 1]\)

\[
\begin{align*}
IMPFS[i] &= \max\{\text{sum}(A[\alpha_i : j]) : j \in [\alpha_i .. \beta_i]\} \\
IS[i] &= \text{sum}(I_i) \\
INXTMS[i] &= \max\{\mathcal{M}_A(j) : \beta_i < j < n\} \\
IRS[i] &= \max\{\text{sum}(A[\beta_i + 1 : j]) : \beta_i < j < n\} \\
IX^*[i] &= \begin{cases} 
\text{the greatest } i \in [\alpha_i .. \beta_i] \text{ such that } A[i] > 0 & \text{if there is such an } i \\
\alpha_i & \text{otherwise}
\end{cases}
\end{align*}
\]

greater than \( n \) be taken into account in the preprocessing step, and similarly to avoid that subsequences of size greater than \( n + 1 \) be considered in the query-answering step. It is therefore not clear whether this idea leads to algorithms for the circular case that are simpler than and at least as efficient as ours.

### 3 Preprocessing Algorithm for the Noncircular Case

We now give a complete description of the preprocessing algorithm for the noncircular case, which we began to introduce in Subsection 2.1. Clearly, if \( n = |A| = 0 \), then nothing needs to be computed in the preprocessing step. If \( n > 0 \), then the auxiliary data that must be computed by the preprocessing algorithm is the interval partition of \( A \) – that is, the number of intervals \( \ell \) and, for all \( i \in [0 .. \ell - 1] \), also \( \alpha_i \) and \( \beta_i \) – and the arrays defined in Table 1. Of all this data, Kadane’s algorithm, which runs in \( O(n) \) time, can easily be made to compute the interval partition of \( A \) and arrays \( K, MS, PRVMS, IMPFS, IS \) and \( IX^* \). Moreover, array \( SFMS \) (which is defined only for \( p < n \)) and array \( INXTMS \) (which is defined only for \( i < \ell - 1 \)) can easily be computed by means of a single right-to-left traversal of array \( MS \) by taking the interval partition of \( A \) into account. Our task in the rest of this section is then to show that the remaining arrays of Table 1 can also be computed in \( O(n) \) time.

With regard to array \( IRS \) (which is defined only for \( i < \ell - 1 \)), note that, for any \( i \in [0 .. \ell - 2] \), if \( IRS[i] = \text{sum}(A[\beta_i + 1 : j]) \) and \( j \in [\alpha' .. \beta' \iota] \), with \( \iota' > i \), then, by the definitions of arrays \( IRS \) and \( IMPFS \), \( IRS[i] = (\sum_{x=i+1}^{\iota'-1} \text{sum}(I_x)) + IMPFS[\iota'] \). Thus, for all \( i \in [0 .. \ell - 2] \),

\[
IRS[i] = \max_{i < \iota' < \ell} \left( \sum_{x=i+1}^{\iota'-1} \text{sum}(I_x) \right) + IMPFS[\iota'],
\]
Algorithm 1: Computation of arrays $X_1$, $X_2$, $BX_1$ and $BX_2$.

1. for $k$ from 0 to $\ell - 1$
   2. if $IX^*[k] > \alpha_k$
      3. $p \leftarrow IX^*[k] - 1$; $X_1[p], X_2[p] \leftarrow IX^*[k]$; $BX_1[p], BX_2[p] \leftarrow MS[p]$
   4. for $p$ from $IX^*[k] - 2$ down to $\alpha_k$
      5. if $MS[p] > MS[X_1[p + 1]]$
         6. $X_1[p] \leftarrow p$; $BX_1[p] \leftarrow MS[X_1[p]]$
         7. $X_2[p] \leftarrow X_2[p + 1]$; $BX_2[p] \leftarrow BX_2[p + 1]$
      8. else
         9. $X_1[p] \leftarrow X_1[p + 1]$; $BX_1[p] \leftarrow \min\{MS[p], BX_1[p + 1]\}$
         10. $X_2[p] \leftarrow X_2[p + 1]$; $BX_2[p] \leftarrow \min\{MS[p], BX_2[p + 1]\}$
      11. if $MS[X_1[p]] - BX_1[p] \geq MS[X_2[p]] - BX_2[p]$
         12. $X_2[p] \leftarrow X_1[p]$; $BX_2[p] \leftarrow BX_1[p]$

which implies that $IRS[i]$ equals $IMPFS[i + 1]$, if $i = \ell - 2$, or $\max\{IMPFS[i + 1], \sum(I_{i+1}) + IRS[i + 1]\}$, if $i < \ell - 2$. We can therefore compute array $IRS$ backwards in $O(\ell) = O(n)$ time by means of arrays $IMPFS$ and $IS$.

The remaining arrays are $X_1$, $X_2$, $BX_2$ (all of which are defined only for $p < IX^*[k]$) and $BX_1$ (which is defined only for $p < X_1[p]$), and they are computed by Alg. 1, which, for every interval $I_k$ of $A$, computes the corresponding stretches of these arrays in $O(I_k)$ time. Algorithm 1 therefore runs in $O(n)$ time. Note that, for any interval $I_k$ such that $IX^*[k] > \alpha_k$, by definition of $IX^*[k]$, $MS[IX^*[k] - 1] < MS[IX^*[k]]$ and $MS[IX^*[k]] \geq MS[i]$ for all $i \in [IX^*[k] + 1 .. \beta_k]$; this directly implies the backwards computation of arrays $X_1$ and $BX_1$ in Alg. 1. The computation of array $BX_2$ is also trivially correct, since it simply accompanies the updates in $p$ and $X_2[p]$. The really interesting feature of Alg. 1 is then its computation of array $X_2$, whose properties are described below.

**Lemma 1.** Let $B_i^p = \min\{MS_A(j) : p \leq j < i\}$ for all $p \in [0 .. n - 1]$ and $i \in [p + 1 .. \beta_k]$. Then the following statements are invariants of the inner loop of Alg. 1:

1. $X_1[p] \leq X_2[p] \leq IX^*[k], p < X_2[p]$ and $BX_2[p] < MS[X_2[p]]$.
2. If $p < X_1[p] < X_2[p]$, then $MS[X_1[p]] - BX_1[p] < MS[X_2[p]] - BX_2[p]$.
3. $MS[X_2[p]] - B_{X_2[p]} \geq MS[i] - B_i^p$ for all $i \in [p + 1 .. \beta_k]$.

**Proof.** We only discuss the last statement, since the other ones are straightforward. First of all, note that $B_{X_2[p]} = BX_2[p]$. Now consider some iteration $k \in [0 .. \ell - 1]$ of the outer loop. To see that Statement 3 is true immediately before the inner loop begins, first note that $p = IX^*[k] - 1$ and $X_2[p] = IX^*[k]$, and then let $i \in [p + 1 .. \beta_k]$. If $i = p + 1 = X_2[p]$, then the statement is trivially true. If $i > X_2[p]$, there are two cases. If $B_i^p = B_{X_2[p]}$, since $MS[X_2[p]] = MS[IX^*[k]] \geq MS[i]$, then the statement is true. Finally, if $B_i^p < B_{X_2[p]}$, since $A[j] \leq 0$ for all $j \in [IX^*[k] + 1 .. \beta_k]$, then $B_i^p = MS[j] \geq MS[i]$ for some $j \in [IX^*[k] + 1 .. i - 1]$, which implies Statement 3.

Now suppose that the statement is valid immediately before iteration $p$, that is, $MS[X_2[p + 1]] - B_{X_2[p + 1]} \geq MS[i] - B_i^p + 1$ for all $i \in [p + 2 .. \beta_k]$; we must show that it also holds at the end of the iteration, that is, that $MS[X_2[p]] - B_{X_2[p]} \geq MS[i] - B_i^p$ for all $i \in [p + 1 .. \beta_k]$. Consider then some $i \in [p + 1 .. \beta_k]$. Note that the algorithm is such that exactly one of the following cases occurs: (1) $X_1[p] = X_1[p + 1]$ and $X_2[p] = X_2[p + 1]$; (2) $X_1[p] \neq X_1[p + 1]$ and $X_2[p] = X_2[p + 1]$; (3) $X_1[p] = X_1[p + 1]$ and $X_2[p] \neq X_2[p + 1]$. The first two cases are simpler and we omit them for brevity. Suppose therefore that $X_2[p] \neq X_2[p + 1]$. Then, $X_2[p] = X_1[p] = X_1[p + 1]$ and $MS[X_1[p + 1]] - B_{X_1[p + 1]} \geq MS[X_2[p + 1]] - B_{X_2[p + 1]}$. We also argue that $B_{X_2[p]} = MS[p]$, which is immediate if $X_1[p + 1] = p + 1$. If $X_1[p + 1] > p + 1$, then, since $X_1[p + 1] \neq X_2[p + 1]$, by
Statement 2, $MS[X1[p+1]] - B_{X1[p+1]}^{p+1} < MS[X2[p+1]] - B_{X2[p+1]}^{p+1}$; this implies $B_{X1[p+1]}^p \neq B_{X1[p+1]}^{p+1}$ and thus $B_{X2[p]}^p = B_{X1[p+1]}^p = MS[p]$. To conclude the proof, there are two cases. If $B_i^p = MS[p]$, since $i \geq p + 1$, then $MS[X2[p]] = MS[X1[p+1]] \geq MS[i]$ and the result follows. If $B_i^p \neq MS[p]$, then $i > p + 1$ and $MS[i] - B_i^p = MS[i] - B_i^{p+1} \leq MS[X2[p+1]] - B_i^{p+1} \leq MS[X2[p+1]] - B_{X2[p+1]}^p \leq MS[X1[p+1]] - B_{X1[p+1]}^p = MS[X2[p]] - B_{X2[p]}^p$, as desired.

We conclude that the preprocessing algorithm for the noncircular case takes $O(n)$ time, as desired.

## 4 Query-answering Algorithm for the Noncircular Case

Given $x$ and $p$, we now show how to compute $MS(A(p))$ in $O(1)$ time given that the auxiliary data described in Section 3 has already been computed. Note that, in case $n = \lvert A \rvert = 0$, then $MS(A(p)) = \max(0, x)$. If $n > 0$, then $MS(A(p))$ is the maximum between zero, $MS(A(p))$, and $MS(A(i))$ for all $i \in [0..n-1]$. Clearly, $MS_{A(p)}(p) = x$, if $p = 0$, and $MS_{A(p)}(p) = x + \max(0, MS_{A}(p-1))$, if $p > 0$. Thus, taking (1) from Subsection 2.1 into account, what remains to be shown is how to compute the preprocessing algorithm for the noncircular case takes $O(n)$ time, as desired.

### 4.1 Handling a Nonnegative $x$

If $x \geq 0$, then, taking (2) from Subsection 2.1 into account, it only remains to compute $\max\{MS_{A(p)}(i') : \beta_k < i < n\}$. In this regard, an important observation is that, although the maximal sums at the elements of $A$ to the right of $I_k$ may increase by different amounts with the insertion of $x$ into $A$, they do so in a regular manner.

More precisely, the maximal sums at interval $I_{k+1}$ increase by $\max(0, x + \sum(I_k))$, and, for all $i \in [k+1..\ell-2]$, if the maximal sums at interval $I_i$ increase by some value $y$, then the maximal sums at interval $I_{i+1}$ increase by $\max(0, y + \sum(I_i))$. By exploiting this regularity, we get the following result:

**Lemma 2.** If $x \geq 0$ and $k < \ell - 1$, then

$$\max\{MS_{A(p)}(i') : \beta_k < i < n\} = \max\{\text{INXTMS}[k], x + IS[k] + IRS[k]\}.$$  

**Proof.** Let us first show that, for all $i \in [\beta_k + 1..n-1]$, $MS_{A(p)}(i') = \max\{MS_{A}(i), sum(A[p][\alpha_k : i'])\}$. Indeed, if $MS_{A(p)}(i') \neq MS_{A}(i)$, then there must be some $j \in [0..p]$ such that $\text{sum}(A[p][j : i']) > \text{sum}(A[p][j : i])$, a contradiction with our choice of $j$. Thus, $j \in [\alpha_k .. p]$. Note also that it cannot be the case that $\text{sum}(A[p][\alpha_k : j - 1]) > 0$, since this again implies $\text{sum}(A[p][\alpha_k : i'']) > \text{sum}(A[p][j : i'])$. Thus, $\text{sum}(A[p][\alpha_k : j - 1]) \leq 0$. Now, either $\alpha_k = j$, which implies that $\text{sum}(A[p][j : i']) = \text{sum}(A[p][\alpha_k : i'])$, or $\alpha_k < j$, which, since $A(p)[\alpha_k : j - 1]$ is a proper prefix of $I_k$, implies that $\text{sum}(A[p][\alpha_k : j - 1]) = 0$ and again that $\text{sum}(A[p][j : i']) = \text{sum}(A[p][\alpha_k : i'])$, as desired.

The previous paragraph implies that the statement of the lemma holds when we substitute the inequality sign “$\geq$” in it. To see that the converse inequality (“$\leq$”) also holds, first note that there are $i, i' \in [\beta_k+1..n-1]$ such that $MS_{A}(i) = \text{INXTMS}[k]$ and $\text{sum}(A[\beta_k+1 : i']) = IRS[k]$. Now, since $x \geq 0$, then $MS_{A(p)}(i') \geq MS_{A}(i) = \text{INXTMS}[k]$. Moreover, $MS_{A(p)}(i') \geq \text{sum}(A[p][\alpha_k : i']) = x + \sum(I_k) + IRS[k]$. Therefore, the second inequality also holds, and so does the lemma.

### 4.2 Handling a Negative $x$

If $x < 0$, then inserting $x$ into interval $I_k$ clearly does not change the maximal sums at elements to the right of $I_k$, that is, $MS_{A(p)}(i') = MS_{A}(i)$ for all $i \in [\beta_k + 1..n-1]$. Thus, if $k < \ell - 1$, then

$$\max\{MS_{A(p)}(i') : \beta_k < i < n\} = \text{INXTMS}[k].$$
Computing $\max\{\mathcal{MS}_{A[i]}(i^o) : p \leq i \leq \beta_k\}$ is more complicated, however, because the maximal sums at elements $A[p], \ldots, A[\beta_k]$ may decrease by different amounts. More precisely, the maximal sum at element $A[p]$ decreases by $\text{dec}(p) = 0$, if $p = \alpha_k$, or by $\text{dec}(p) = \min\{-x, \mathcal{MS}_{A}(p-1)\}$, if $p > \alpha_k$, and, for all $i \in [p .. \beta_k - 1]$, if the maximal sum at element $A[i]$ decreases by $\text{dec}(i)$, then the maximal sum at element $A[i + 1]$ decreases by $\text{dec}(i + 1) = \min\{\text{dec}(i), \mathcal{MS}_{A}(i)\} = \min\{\text{dec}(p), B_{i+1}^{p}\}$ (recall that term $B_{i+1}^{p}$ is defined in Lemma 1). Thus, for all $i \in [p .. \beta_k]$,

$$\mathcal{MS}_{A[i]}(i^o) = \mathcal{MS}_{A}(i) - \text{dec}(i).$$

The key to compute $\max\{\mathcal{MS}_{A[i]}(i^o) : p \leq i \leq \beta_k\}$ quickly is to realize that it is not necessary to compute $\mathcal{MS}_{A[i]}(i^o)$ explicitly for every index $i \in [p .. \beta_k]$. Indeed, a careful analysis of the problem shows that we do not need to consider more than two such values of $i$:

**Lemma 3.** If $x < 0$ and $p < n$, letting $x_1 = X1[p]$, $x_2 = X2[p]$ and $x^* = IX^*[k]$, then $\max\{\mathcal{MS}_{A[i]}(i^o) : p \leq i \leq \beta_k\}$ equals

$$\begin{cases} 
\mathcal{MS}_{A[p]}(p^o), & \text{if } p \geq x^* \text{ and } \mathcal{MS}_{A[p]}(p^o) \geq 0; \\
0 \text{ or less}, & \text{if } p \geq x^* \text{ and } \mathcal{MS}_{A[p]}(p^o) < 0; \\
\max\{\mathcal{MS}_{A[i]}(x_1^o), \mathcal{MS}_{A[i]}(x_2^o)\}, & \text{if } p < x^*. 
\end{cases}$$

**Proof.** If $p \geq x^*$, then note that, by definition of $x^*$, $\mathcal{A}[i] \leq 0$ for all $i \in [x^* + 1 .. \beta_k]$. Thus, for all $i \in [p + 1 .. \beta_k]$, if $\mathcal{MS}_{A[p]}(p^o) \geq 0$, then $\mathcal{MS}_{A[i]}(i^o) \leq \mathcal{MS}_{A[p]}(p^o)$, and, if $\mathcal{MS}_{A[p]}(p^o) < 0$, then $\mathcal{MS}_{A[i]}(i^o) = \mathcal{A}[i] \leq 0$, which implies the first two cases of (4).

Now suppose that $p < x^*$ and that $\mathcal{MS}_{A[p]}(i^o) > \mathcal{MS}_{A[i]}(x_1^o)$ for some $i \in [p .. \beta_k]$. Since, by definition of $x_1$, $\mathcal{MS}_{A[i]}(i) \leq \mathcal{MS}_{A}(x_1)$, then, by (3), $\text{dec}(i) < \text{dec}(x_1)$. Hence, by definition of $\text{dec}(i)$, $x_1 < i$, which implies that $\text{dec}(i) = \min\{\text{dec}(x_1), B_{\beta_k}^p\}$ and thus that $\text{dec}(i) = B_{\beta_k}^p$. Therefore, by (3) and Statement 3 of Lemma 1, $\mathcal{MS}_{A[p]}(i^o) = \mathcal{MS}_{A}(i) - B_{\beta_k}^p \leq \mathcal{MS}_{A}(x_2) - B_{\beta_k}^p \leq \mathcal{MS}_{A[i]}(x_2^o)$. Since, by Lemma 1, both $x_1$ and $x_2$ belong to $[p .. \beta_k]$, then the last case of (4) also holds.

By Lemma 3 and (3), since our algorithm has at its disposal arrays $IX^*$, $\mathcal{M}$, $X1$, $X2$, $BX1$ and $BX2$, then it can compute $\max\{\mathcal{MS}_{A[i]}(i^o) : p \leq i \leq \beta_k\}$ in $O(1)$ time if $p < n$, as desired. (Note that, in the second case of (4), we do not know the exact value of $\max\{\mathcal{MS}_{A[i]}(i^o) : p \leq i \leq \beta_k\}$, but, since we know it is nonpositive, then we can ignore it for the purpose of computing $\mathcal{MS}(A[p])$.)

Gathering the results of this section, we get:

**Theorem 1.** Provided that the $O(n)$ time preprocessing algorithm of Section 3 has already been executed on $A$, our query-answering algorithm computes $\mathcal{MS}(A[x-p])$ in $O(1)$ worst-case time for any $x \in \mathbb{R}$ and $p \in [0 .. n]$.

## 5 Preliminaries for the Circular Case

A few extra definitions will be used in the circular case. For any $m \in \mathbb{N}\setminus\{0\}$, $x \in [0 .. m-1]$ and $y \in \mathbb{Z}$, let $x + [m] y$ be defined as $(x + y) \mod m$, if $y \geq 0$, or as the $z \in [0 .. m - 1]$ such that $z + [m] y = x$, if $y < 0$. Moreover, for any $m \in \mathbb{N}\setminus\{0\}$ and $a,b \in [0 .. m-1]$, let $[a : b] = [a .. b]$, if $a \leq b$, and $[a : b] = \{i \in \mathbb{N} : a \leq i < m \text{ or } 0 \leq i \leq b\}$, if $b < a$. The first notation is for “sum modulo $m$” and is used to circularly traverse the indices of a sequence $S$ of size $m$; the complementary operation “subtraction modulo $m$” is defined as $x - [m] y = x + [m] (-y)$. The second notation is for “circular ranges”: in particular, $[a : b]$ is the set of the indices of the elements of $S[a : b]$. (Making “$m$” explicit is necessary in both notations, because we use them in contexts where sequences of different sizes are considered.)

As explained in Subsection 2.2, the circular case mainly departs from the noncircular case because of the so-called “type 3” sequences, such as the one depicted in Fig. 3. In that sequence, elements $A[12]$, $A[13]$, $A[0]$, $A[1]$ and $A[2]$ should intuitively belong to the same interval, since $\mathcal{MCS}_{A}(i + [n] 1) = \mathcal{MCS}_{A}(i) + A[i + [n] 1]$ for all $i \in [12 .. 1]$. On the other hand, $A[11]$ should not belong to the same
interval as $A[12]$, since $MCS_A(12) \neq MCS_A(11) + A[12]$. From these considerations, and taking our knowledge of the noncircular case into account, one would then surmise that, for all $i \in [12 \div 2]$, $MCS_A(i) = \sum(A[12 : i])$; however, in reality we have $MCS_A(i) = \sum(A[2+1 : i]) > \sum(A[12 : i])$.

As we show later (Observation 3), this happens because, although $A$ has negative numbers, it is such that $MCS_A(i) \geq 0$ for all $i$; in particular, and in contrast to the noncircular case, this implies that an interval cannot end at some element $A[i]$ such that $MCS_A(i) < 0$. What is then common between the circular and noncircular cases is that, whenever two elements $A[i]$ and $A[j]$ belong to the same interval $I$, the maximal (circular) sums at these elements equal the sums of subsequences of $A$ which begin at a common element $A[h]$ — which, however, may not be the first element of $I$.

The discussion above indicates that, for each element $A[i]$ of $A$, it is important to know an “origin” element $A[h]$ such that $MCS_A(i) = \sum(A[h : i])$. We therefore define $OMS$ as the array such that $OMS[i] = i - [n] \max[j \in [0 .. n-1] : \sum(A[i - [n] j : i]) = MCS_A(i)]$ for all $i \in [0 .. n-1]$. Intuitively, $OMS[i]$ is, in relation to $i$, the circularly leftmost index $j \in [0 .. n-1]$ such that $\sum(A[j : i]) = MCS_A(i)$. Clearly, the following holds:

**Observation 1.** For all $i \in [0 .. n-1]$, $\sum(A[OMS[i] : i]) = MCS_A(i)$; moreover, if $OMS[i] \neq i + [n] 1$, then every suffix of $A[i + [n] 1 : i]$ of $OMS[i] - [n] 1$ has negative sum.

Our definition of interval for the circular case is then that two elements $A[i]$ and $A[j]$ belong to the same interval iff $OMS[i] = OMCS[j]$. However, as is the case in Fig. 3, this may lead to a “broken” interval in $A$, made up of a suffix and a prefix of the sequence. Since it is convenient that all intervals have contiguous elements, we henceforth suppose that $A$ is such that either (1) $OMS[n - 1] \neq OMCS[0]$ or (2) $OMS[i] = 0$ for all $i \in [0 .. n-1]$. Note that, if this is not true of the sequence $A$ that is supplied as input to the preprocessing algorithm, then, as argued in Subsection 2.2, we can simply perform a suitable circular shift on $A$ and take this into account in the query-answering algorithm; Sections 7 and 8 give details on this. Finally, note that our definition of interval immediately implies a definition of interval partition for the circular case: $\alpha_0 = 0, \beta_0 = \max\{i \in [0 .. n-1] : OMCS[i] = OMCS[0]\}, \alpha_1 = \beta_0 + 1$ (if $\beta_0 < n-1$), etc.\(^1\)

We finish this section with some simple but important observations about the “structure” of sequences of types 2 and 3. The next result follows directly from the definition of array $OMCS$ and our supposition about $A$.

**Observation 2.** If $A$ is of type 2, then $MCS_A(n-1) < 0$, and furthermore $MCS_A(i) = MS_A(i)$ for all $i \in [0 .. n-1]$.

Item 1 of the next lemma states the basic property of array $OMCS$ when $A$ is of type 3. Item 2 implies, together with our supposition about $A$, that intervals are always strictly contiguous, and also that, if $i = OMCS[i]$ for some $i \in [0 .. n-1]$, then $OMS[h] = i$ for all $h \in [0 .. n-1]$ (i.e. $A$ has only one interval).

**Lemma 4.** If $A$ is of type 3, the following holds for all $i \in [0 .. n-1]$:

1. If $OMCS[i] \neq i + [n] 1$, then $OMCS[i + [n] 1] = OMCS[i]$.

\(^{1}\)Note that this definition applies to (and agrees with the one used in) the noncircular case as well: we just need to use, instead of array $OMCS$, the array $OMS$ such that $OMS[i] = \min\{j \in [0 .. i] : \sum(A[j : i]) = MS_A(i)\}$.
2. \(OMCS[h] = OMCS[i]\) for all \(h \in [i, n]\). \(OMCS[i] = \sum_{i} A[i+1..n]\).

Proof. Given \(i\), let \(j = i + [n] 1\), \(a = OMCS[i]\) and \(b = OMCS[j]\). With regard to Item 1, if \(a \neq j\), then, by Observation 1 and the definition of a type 3 sequence, \(\sum(A[a \leq i]) = MCS_A(i) \geq 0\), and thus \(b \neq j\); moreover, since every suffix of \(A[j \leq a] \in [a, n] 1\) has negative sum, then \(b \notin [j, a] \in [a, n] 1\).

Finally, we cannot have \(a \neq i\) and \(b \in [a, n] 1\), because this would imply, by Observation 1, that \(\sum(A[a \leq b] i) < 0\), and thus that \(\sum(A[b \leq i]) > MCS_A(i)\), contradicting the definition of \(MCS_A(i)\). Thus, we conclude that \(b = a\), as desired.

With regard to Item 2, let \(x \in [1 .. n]\) be such that \(a = i + [n] x\). By applying the previous item, a straightforward induction then shows that \(OMCS[h] = a\) for all \(h \in \{i, i + [n] 1, i + [n] 2, \ldots, i + [n] (x - 1)\} = [i, [n] a - [n] 1]\).

The next observation shows that, when \(A\) is of type 3, although its intervals define a partition of the sequence, the maximal circular sum at an element \(A[j]\) of an interval \(I_1\) of \(A\) depends on the elements of all other intervals of \(A\) (as in Fig. 3); this contrasts with the noncircular case, where the maximal sum at \(A[j]\) equals \(\sum(A[a_1 : j])\) and thus only depends on elements of \(I_1\). The result follows easily from Lemma 4, Observation 1 and our supposition about \(A\).

**Observation 3.** If \(A\) is of type 3, then, for every interval \(I_1\) and \(j \in [\alpha, \beta]\), \(OMCS[j] = \beta_i + [n] 1\); moreover, any prefix of \(I_1\) has nonnegative sum.

## 6 The Max-queue Data Structure

Recall the definition of the max-queue data structure from Subsection 1.4: Algorithm 2 provides a simple implementation of this data structure in an object-oriented pseudocode. (In the algorithm, “\(\text{c} \lor \text{d} \)” denotes the “conditional or” Boolean operator, which evaluates its right operand iff the left one is false.) The algorithm provides five methods, which implement, in order, the operations of (1) initialization, (2) peek of greatest key, (3) peek of the satellite data of the oldest element with greatest key, (4) removal and (5) insertion. Method \(\text{Push}\) adds its argument \(d\) to all already stored keys and then inserts a new key \(v\) and its associated satellite data \(sd\) in the data structure. (The type \(T\) of \(sd\) is supplied as argument to method \(\text{InitMaxQueue}\) during initialization.) Methods \(\text{Pop}\), \(\text{GetKeyMax}\) and \(\text{GetSatDataMax}\) may be called iff the data structure is not empty.

The fundamental ideas behind Alg. 2 are two. First, in order that operations “peek” and “pop” execute quickly, we do not store all inserted keys explicitly: instead, we maintain a deque \(Q\) which only stores the maximal key, the second maximal, the third maximal, etc. (whenever two elements have the same key, we store both\(^2\)); thus, we always know that the maximal key of the data structure is the last element of \(Q\), and, when this element is removed, we know that the new maximum is the new last element of \(Q\). Note that removing the keys that according to this criterion should not be stored in \(Q\) can be efficiently done on the occasion of each insertion: if \(\langle v_0, v_1, \ldots, v_{n-1} \rangle\) denotes a deque such that \(v_0 \leq v_1 \leq \ldots \leq v_{n-1}\) (a property which is trivially true when the deque is initialized, since \(n = 0\)), and if \(v\) and \(d\) are the arguments to an insertion operation, then the deque that results from this insertion is either \(\langle v \rangle\), if \(n = 0\) or else if \(v > v_{n-1} + d\), or \(\langle v, v, v, v, v, v, \ldots, v, v, v, v, v, v, v, d \rangle\), where \(i = \min\{j \in [0 .. n-1] : v \leq v_j + d\}\), if \(v \leq v_{n-1} + d\); clearly, then, we should remove from \(Q\) every \(v_j\) such that \(v > v_j + d\), which is exactly what is done in method \(\text{Push}\). Moreover, we keep track of the number of elements implicitly stored between any given keys \(v_j\) and \(v_{j+1}\) by storing this number in a field “implicit” associated with \(v_j\).

The second fundamental idea behind Alg. 2 is to avoid updating the keys of all elements of \(Q\) on every insertion, and instead to do it implicitly. More precisely, we define an object attribute \(\text{max}\), whose value is defined as the greatest key stored in the data structure (which is the only key we ever need to know at any moment). Note that updating \(\text{max}\) is trivial on any call \(\text{Push}(v, d, sd):\) we must only add \(d\) to it, unless \(v\) becomes the only key of \(Q\), in which case \(\text{max}\) must be set to \(v\). Finally,

---

\(^2\)We do so because it is useful for our purposes in this paper, but, in the cases where this behaviour is not necessary, it makes sense to store only the most recently inserted element, since this reduces the size of the data structure.
Algorithm 2: Class implementing the max-queue data structure.

```plaintext
1 attribute max : R
2 attribute Q : deque of record {orig_value : R, diff : R, implicit : N, sat_data : T}
3 method InitMaxQueue(T) : no_return begin Q.InitEmptyDeque() end
4 method GetKeyMax() : R begin return max end
5 method GetSatDataMax() : T begin return Q.last.sat_data end
6 method Pop() : no_return
7 begin
8     if Q.last.implicit > 0 then Q.last.implicit ← Q.last.implicit − 1
9     else
10        acc_d ← max − Q.last.orig_value; Q.PopBack()
11        if not Q.IsEmpty() then max ← Q.last.orig_value + acc_d − Q.last.diff
12 method Push(v : R, d : R, sd : T) : no_return
13 begin
14     impl ← 0; acc_d ← d; continue ← true
15     while continue = true do
16         if Q.IsEmpty() c_or Q.first.orig_value + acc_d ≥ v then
17             continue ← false
18         else
19             acc_d ← acc_d + Q.first.diff; impl ← impl + 1 + Q.first.implicit; Q.PopFront()
20         if Q.IsEmpty() then max ← v else max ← max + d
21         Q.PushFront({v, acc_d, impl, sd})
```

to see how to update max on a call Pop(), first note that max must only be updated if there is no implicitly stored element that is older than the last element of Q, and if Q has at least two elements (otherwise it will be empty after the call to Pop). In this case, if Push(v, d, sd) denotes the i-th most recent call to method Push, with i ≥ 0, and if v_i and v_k are respectively the current last and second to last elements of Q (thus 0 ≤ k < l), then it currently holds that max = v_l + ∑_{i=0}^{l-1} d_i, and the new value of max (after the call to Pop) must be v_k plus ∑_{i=0}^{k-1} d_i = max − v_l + ∑_{i=k}^{l-1} d_i. For us to obtain the value of the last summation efficiently, in Alg. 2 we associate with every key v_k of Q the value diff, defined as follows: if (v_i, v_i, . . . , v_{i−n}) is the sequence of keys of Q, with v_{i−n} = v_k, then, for all j ∈ [0 .. n − 2], diff_j = ∑_{i=j}^{j−1} d_i. Note that the value “diff” associated with a given key v is always the same, and that it can be efficiently computed when v is inserted into Q by making use of value diff_j for each key v_j removed during this insertion (see the manipulation of variable acc_d in method Push). Thus, since diff_k = ∑_{i=k}^{l−1} d_i, we conclude that our calculation of max in method Pop is correct.

Our previous considerations then imply:

**Theorem 2.** Algorithm 2 is correct, i.e. it implements the max-queue data structure as described in Subsection 1.4; in particular, any set of m operations on a max-queue takes O(m) worst-case time.

**Proof.** The correctness of Alg. 2 follows from our arguments in the paragraphs above. With regard to its run time, first note that, except for Push, all methods trivially run in O(1) worst-case time. Moreover, each call to method Push runs in O(max{1, r}) worst-case time, where r is the number of elements removed from Q during that call. Since in m operations at most m elements are removed from Q, then all calls to Push collectively take O(m) time, which closes the argument.

□
7 Preprocessing Algorithm for the Circular Case

We now describe our preprocessing algorithm for the circular case. The first task of the algorithm is to compute \( \text{sum}(A) \) and to check whether sequence \( A \) is of type 1, which can be done with a single traversal of the sequence. If \( A \) is of type 1, then the algorithm does not need to compute anything else and thus terminates. If, however, \( A \) is not of type 1, then the next step of the algorithm is to compute the array \( \text{MCS} \) such that \( \text{MCS}[i] = \text{MCS}_A(i) \) for all \( i \in [0 \ldots n-1] \), as follows. The algorithm begins with an initially empty max-queue and then performs \( n \) insertions: for all \( i \in [0 \ldots n-1] \), insertion \( i \) first adds \( A[i] \) to all already inserted keys and then enqueues \( A[i] \), as indicated below:

\[
\begin{align*}
&0 \rightarrow A[0] \rightarrow \left[ \sum_{i=0}^{1} A[i], A[1] \right] \rightarrow \left[ \sum_{i=0}^{2} A[i], \sum_{i=1}^{2} A[i], A[2] \right] \rightarrow \left[ \sum_{i=0}^{n-1} A[i], \sum_{i=1}^{n-1} A[i], \sum_{i=2}^{n-1} A[i], \ldots, \sum_{i=n-2}^{n-1} A[i], A[n-1] \right].
\end{align*}
\]

Clearly, the greatest key of the max-queue that results from these \( n \) operations equals \( \text{MCS}_A(n-1) \), so the preprocessing algorithm peeks this value and sets \( \text{MCS}[n-1] \) to it. To obtain the remaining elements of array \( \text{MCS} \), the algorithm performs the following operations on the max-queue: for all \( i \in [0 \ldots n-2] \), (1) remove the oldest element (“pop”), (2) add \( A[i] \) to all keys, (3) enqueue \( A[i] \), and (4) peek the greatest key and set \( \text{MCS}[i] \) to it. As an example, note that the state of the max-queue after iteration \( i = 0 \) is

\[
\left[ \left( \sum_{i=1}^{n-1} A[i] \right) + A[0], \left( \sum_{i=2}^{n-1} A[i] \right) + A[0], \ldots, \left( \sum_{i=n-2}^{n-1} A[i] \right) + A[0], A[n-1] + A[0], A[0] \right],
\]

and that the greatest key of the max-queue in this state is actually \( \text{MCS}_A(0) \). We conclude that array \( \text{MCS} \) can be computed by means of \( O(n) \) max-queue operations and, by Theorem 2, that this takes \( O(n) \) time. To finalize the discussion of this first part of the preprocessing algorithm, note that, since the “peek” operation of a max-queue always refers to the oldest element with greatest key, we can get array \( \text{OMCS} \) directly as a by-product of our computation of array \( \text{MCS} \). For this purpose, only two additions to the computation above are necessary: (1) supply index \( i \) as satellite data every time \( A[i] \) is enqueued; (2) every time \( \text{MCS}[i] \) is set to the key of the maximum (including when \( i = n-1 \)), set \( \text{OMCS}[i] \) to the satellite data of the maximum.

Now, as discussed in Section 5, we need to make sure that no interval of \( A \) is broken into two parts. Moreover, if \( A \) has only one interval, it is convenient to have \( \text{OMCS}[i] = 0 \) for all \( i \). We achieve both things by circularly shifting sequence \( A \) \( \alpha^c \) positions to the left, where \( \alpha^c = \text{OMCS}(0) \), if \( A \) has only one interval, and \( \alpha^c = \min\{i \in [0 \ldots n-1] : \text{OMCS}[i - [n] 1] \neq \text{OMCS}(i)\} \), otherwise. Moreover, in order that arrays \( \text{MCS} \) and \( \text{OMCS} \) remain consistent with \( A \), we can either compute them from scratch again or perform the same left-shift on them (in the latter case, we also need execute command \( \text{OMCS}[i] \leftarrow \text{OMCS}[i] - [n] \alpha^c \) for every index \( i \)). Clearly, all this can be done in \( O(n) \) time.

The rest of the data that the preprocessing algorithm needs to compute is the following: (1) the type of sequence \( A \) and its interval partition (as defined in Section 5); (2) arrays \( K, \text{IMPS} \) and \( IS \) of Table 1, but all relative to the interval partition induced by array \( \text{OMCS} \), and furthermore array \( K \) having with its indices restricted to range \([0 \ldots n-1]\); (3) arrays \( \text{PRVMS}, \text{INXTMS}, \text{IX}^*, X_1, X_2, BX_1 \) and \( BX_2 \) of Table 1, if \( A \) is of type 2; (4) the arrays of Table 2. Clearly, (1) and (2) can be done in \( O(n) \) time by using sequence \( A \) and arrays \( \text{MCS} \) and \( \text{OMCS} \); moreover, since \( \text{MCS}(A^{(x ightarrow 0)}) = \text{MCS}(A^{(x ightarrow 0)}) \) is true for any \( x \), then the query-answering algorithm can treat an insertion into position \( p = n \) as one into position \( p = 0 \), which implies that, in the circular case, no array needs to have indices in the range \([0 \ldots n]\). Now, if \( A \) is of type 2, then the required arrays of Table 1 can be computed exactly as described in Section 3 (with array \( \text{MCS} \) replacing array \( MS \)): by Observation 2, these arrays are consistent with the ones computed using the interval partition induced by array \( \text{OMCS} \). What remains to be shown is then how to compute the arrays of Table 2. We do not further discuss, however, the computation of arrays \( \text{PFMCS} \) (which is defined only for \( p > \alpha_k \), \( \text{SFMCS} \) and \( \text{IMCS} \), which can easily be computed by simple traversals of array \( \text{MCS} \).
IRCS is therefore analogous to that of array \( A \). We have changing the keys currently stored in the max-queue. For brevity, we omit further details about it.

We first claim that

\[
\text{Proof.}
\]

Lemma 5. If \( A \) is nonnegative; thus, there are \( i, j \in [p \plus{} 1: \alpha_k .. \beta_k + 1] \) such that \( \text{MS}(B) = \sum(S) \), where \( S = A[i \plus{} 1: j] \).

Given such \( i \) and \( j \), either \( S \) encompasses element \( A[\beta_k + [n] 1] \) (that is, \( i \in [p \plus{} 1: \beta_k + [n] 1] \) and \( j \in [\beta_k + [n] 1] \plus{} 1 \plus{} 1 \) \( \alpha_k .. [n] 1 \)) or not. In the second case, either \( S \) is a subsequence of \( A[\beta_k] \), in which case our claim holds, or \( S \) is a subsequence of \( A[1: \beta_k] \), in which case our claim also holds, since,

| Arrays with indices in the range \([0 .. n - 1]\) |
|---|
| \( PFMCS[p] \) | \( = \max\{\text{MCS}_A(i) : \alpha_k \leq i < p\} \) |
| \( PFRMMS[p] \) | \( = \max\{\text{MCS}_A(i) : i \in [0 .. n - 1] \setminus [\alpha_k .. p - 1]\} \) |
| \( PFRMMSS[p] \) | \( = \text{MS}(A[p : \beta_k] + A[1: k]) \) |
| \( PFBS[p] \) | \( = \min\{0\} \cup \{\sum(A[i : j]) : \alpha_k < i \leq j < p\} \) |
| \( SFMCS[p] \) | \( = \max\{\text{MCS}_A(i) : p \leq i \leq \beta_k\} \) |

| Arrays with indices in the range \([0 .. \ell - 1]\) |
|---|
| \( IMCS[i] \) | \( = \max\{\text{MCS}_A(j) : j \in [\alpha_i .. \beta_i]\} \) |
| \( IOMCS[i] \) | \( = \max\{\text{IMCS}[j] : i \neq j \in [0 .. \ell - 1]\} \) |
| \( IRCS[i] \) | \( = \text{the greatest sum of a prefix of } A[1 \plus{} 1] \) |

Table 2: Definitions of extra arrays used in the circular case. Those indexed with \( p \) (instead of \( i \)) make use of \( k = K[p] \) in their definitions. Some arrays are not defined for every index (see Section 7).

For every \( I_i, A[1 \plus{} 1] = I_{i+1}^1 + I_{i+2}^2 + \ldots + I_{i+\ell-1}(\ell-1) \).

Array \( IRCS \) is the circular equivalent of array \( IRS \) of Table 1; it must be computed iff \( \ell > 1 \). Note that, given some interval \( I_i \), if \( IRCS[i] = \sum(A[\beta_i + [n] 1 : j]) \) and \( j \in [\alpha_i .. \beta_i] \), then certainly \( \sum(A[\alpha_i : j]) = \text{IMPFS}[i] \). It follows that \( IRCS[0] \) equals the greatest key of max-queue

\[
\left[ \sum_{i=1}^{\ell-2} IS[i] + \text{IMPFS}[\ell-1], \ldots, \sum_{i=1}^{2} IS[i] + \text{IMPFS}[3], \sum_{i=1}^{1} IS[i] + \text{IMPFS}[2], \text{IMPFS}[1] \right].
\]

This max-queue may be built from an empty one by \( \ell - 1 \) insertion operations: for all \( i \) from \( \ell - 1 \) to \( 1 \), add \( IS[i] \) to all current keys and then enqueue \( \text{IMPFS}[i] \). Moreover, by removing the oldest element from this max-queue and then performing the insertion operation in question with index \( 0 = (\ell - 1) + [1] \), we get a max-queue whose greatest key equals \( IRCS[\ell - 1] \). By repeating this procedure for indices \( \ell - 2, \ldots, 1 \), we finally obtain array \( IRCS \). Note that the computation of array \( IRCS \) is therefore analogous to that of array \( MCS \), except that it only takes \( O(\ell) = O(n) \) time.

Array \( IOMCS \) must also be computed iff \( \ell > 1 \). Its computation is analogous to that of array \( IRCS \), but is simpler: each insertion operation on the max-queue inserts a new key \( IMCS[i] \) without changing the keys currently stored in the max-queue. For brevity, we omit further details about it.

Array \( PFBS \) is defined only for \( p > \alpha_k \). Note then that, letting \( A^- = (-A[0], -A[1], \ldots, -A[n-1]) \), we have \( \text{PFBS}[p] = -\text{MS}(A^-[\alpha_k + 1 : p - 1]) \) for all \( p \in [0 .. n - 1] \) such that \( p > \alpha_k \). This observation immediately implies that, for every interval \( I_i \) of \( A \), we can compute \( \text{PFBS}[j] \) for all \( j \in [\alpha_i + 1 .. \beta_i] \) by means of a single run of Kadane’s algorithm with input \( A^-[\alpha_i + 1 : \beta_i - 1] \); since this takes \( O(|I_i|) \) time, it follows that we can compute array \( PFBS \) in \( O(n) \) time.

Array \( PFRMMMS \) must be computed iff \( A \) is of type 3. For any \( p \), \( PFRMMMS[p] = \text{MS}(A[p : \beta_k] + A[1: k]) \), where \( A[I_i] = I_{i+1}^1 + I_{i+2}^2 + \ldots + I_{i+\ell-1}(\ell-1) \) for every interval \( I_i \). Thus, if \( \ell = 1 \), then \( A[1] = \emptyset \) and array \( PFRMMMS \) can be computed by means of a right-to-left run of Kadane’s algorithm on \( A \). If \( \ell > 1 \), we have:

Lemma 5. If \( \ell > 1 \), then \( PFRMMMS[p] = \text{MS}(A[p : \beta_k] + \langle IRCS[k] \rangle) \) for all \( p \in [0 .. n - 1] \).

Proof. We first claim that \( PFRMMMS[p] \leq \text{MS}(A[p : \beta_k] + \langle IRCS[k] \rangle) \). Indeed, letting \( B = A[p : \beta_k] + A[1: k] \), \( PFRMMMS[p] \) is defined as \( \text{MS}(B) \). Now, from Observation 3, the first element of \( I_{k+1} \) is nonnegative; thus, there are \( i, j \in [p \plus{} 1: \alpha_k .. [n] 1] \) such that \( \text{MS}(B) = \sum(S) \), where \( S = A[i \plus{} 1: j] \).

Given such \( i \) and \( j \), either \( S \) encompasses element \( A[\beta_k + [n] 1] \) (that is, \( i \in [p \plus{} 1: \beta_k + [n] 1] \) and \( j \in [\beta_k + [n] 1] \plus{} 1 \plus{} 1 \) \( \alpha_k .. [n] 1 \)) or not. In the second case, either \( S \) is a subsequence of \( A[\beta_k] \), in which case our claim holds, or \( S \) is a subsequence of \( A[1: \beta_k] \), in which case our claim also holds, since,
by Observation 3, the sum of any subsequence of $A^{I_k}$ is not greater than $IRCS[k]$. In the first case, $S = S' + A[β_k + [n .. j]]$, where $S'$ equals either $\langle i \rangle$, if $i \notin [p .. β_k)$, or $A[i : β_k]$, otherwise; in this case, since $\text{sum}(A[β_k + [n .. j]]) \leq IRCS[k]$, our claim is again proved.

Finally, to see that $\text{PFRRMM}[p] \geq \text{MS}(A[p : β_k] + \langle IRCS[k] \rangle)$, note that, for every maximal sum subsequence $S$ of $A[p : β_k] + \langle IRCS[k] \rangle$, it is easy to find a subsequence $S'$ of $A[p : β_k] + A^{I_k}$ such that $\text{sum}(S') = \text{sum}(S)$.

Thus, for every interval $I_i$ of $A$, we can compute $\text{PFRRMM}[j]$ for all $j \in [α_i .. β_i]$ by means of a single right-to-left run of Kadane’s algorithm on sequence $I_i + \langle IRCS[i] \rangle$; since this takes $O(|I_i| + 1)$ time, we can compute array $\text{PFRRMM}$ in $O(n + \ell) = O(n)$ time.

Array $\text{PFRRMMCS}$ must also be computed iff $A$ is of type 3. Its definition immediately implies that, for all $p \in [0 .. n - 1]$, $\text{PFRRMMCS}[p]$ equals $\text{MCS}[p]$, if $p = β_k$ and $\ell = 1$, or $\max\{\text{MCS}[p], \text{IOMCS}[k]\}$, if $p = β_k$ and $\ell > 1$, or $\max\{\text{MCS}[p], \text{PFRRMMCS}[p + 1]\}$, if $p < β_k$. This directly implies a right-to-left computation of the array in $O(n)$ time.

We conclude that the preprocessing algorithm for the circular case takes $O(n)$ time, as desired.

8 Query-answering Algorithm for the Circular Case

We now show how to compute $\text{MCS}(A^{(p)})$ in $O(1)$ time given that the auxiliary data described in Section 7 has already been computed. It is easy to see that $\text{MCS}(A^{(p)}) = \max\{0, \text{sum}(A)\} + \max\{0, x\}$ if $A$ is of type 1. If $A$ is not of type 1, then first note that $\text{MCS}(A^{(x \rightarrow n)}) = \text{MCS}(A^{(x \rightarrow 0)})$; thus, if $p = n$, we for convenience set $p$ to zero. Moreover, since the preprocessing step shifts the original sequence $A$ $α^c$ positions to the left, we set $p$ to $p - [n .. 1]$. The rest of our treatment depends both on the sign of $x$ and on the type of sequence $A$.

8.1 Handling a Nonnegative $x$ and a Sequence of Type 2

If $A$ is of type 2, then we know from Observation 2 that $\text{MCS}_A(i) = \text{MS}_A(i)$ for all $i$. Inserting a number $x \geq 0$ into $A$ then affects the sequence similarly as in the noncircular case: roughly speaking, if the maximal circular sums in an interval $I_y$ increase by some value $z$, then the maximal circular sums in the “next” interval increase by $\max\{0, z + \text{sum}(I_y)\}$. The essential difference in this case is that also interval $I_{ℓ-1}$ has a “next” interval, namely $I_0$. This is so because, as exemplified in Fig. 2, the insertion of $x$ may also change the maximal circular sum at an element $A[i]$ such that $i < p$, even if $i \in [α_k .. p - 1]$. We therefore need an extension of our results for the noncircular case, which is provided below:

Lemma 6. If $A$ is of type 2 and $x \geq 0$, then $\text{MCS}_{A^{(p)}}(p) = x + \max\{0, \text{MCS}_A(p - [n .. 1])\}$. Moreover, for all $i \in [0 .. n - 1]$, $\text{MCS}_{A^{(p)}}(i^c)$ equals:

1. $\text{MCS}_A(i) + x$, if $i \in [p .. β_k]$.
2. $\max\{\text{MCS}_A(i), x + \text{sum}(A[α_k .. i])\}$, if $i \notin [α_k .. β_k]$.
3. $\max\{\text{MCS}_A(i), x + \text{sum}(A) - \text{sum}(A[i + 1 : j - 1]) : j \in [i + 1 .. p]\}$, if $i \in [α_k .. p - 1]$.

Proof. We only discuss the last two items, since the others are straightforward. Suppose that $i \in [0 .. n - 1] \setminus [p .. β_k]$ and let $I_{ℓ'}$ be the interval of $A$ such that $i \in [α_{ℓ'} .. β_{ℓ'}]$. Since $A[α_{ℓ'} : i] = A^{(p)}[(α_{ℓ'})^c : i^c]$, then $\text{MCS}_{A^{(p)}}(i^c) \geq \text{MCS}_A(i)$. Now suppose that $\text{MCS}_{A^{(p)}}(i^c) > \text{MCS}_A(i)$ (otherwise the result is proved). Then $\text{MCS}_{A^{(p)}}(i^c) = \text{sum}(S)$ for some subsequence $S$ of $A^{(p)}$ which ends in $A[i^c]$ and includes $x$. Thus, letting $s$ be the greatest sum of such a subsequence $S$, we have $s \geq \text{MCS}_{A^{(p)}}(i^c)$. Now, if $k \neq k'$, since intervals have negative suffixes and nonnegative proper prefixes, then $s = \text{sum}(A^{(p)}[α_k .. i^c]) = x + \text{sum}(A[α_k .. i])$. On the other hand, if $k = k'$, then $i \in [α_k .. p - 1]$ and $s$ equals the greatest sum of a suffix of $A^{(p)}[i + 1 : i^c]$, which includes $x$, that is, $s = \text{sum}(A^{(p)}[j : i^c]) = x + \text{sum}(A[i + 1 : j - 1])$ for some $j \in [i + 1 .. p]$. Finally, since $s$ is also a lower bound on $\text{MCS}_{A^{(p)}}(i^c)$, then the result is proved.

□
The lemma above and the definitions of the arrays computed in Section 7 immediately imply:

**Corollary 1.** If $A$ is of type 2 and $x \geq 0$, then $\text{MCS}(A^{(p)})$ is the maximum among \(1 \) $x + \max\{0, \text{MCS}[p - [n]]\}$, \(2 \) $x + \text{SFMCS}[p]$, \(3 \) $\max\{\text{IOMCS}[k], x + \text{IS}[k] + \text{IRCS}[k]\}$, and \(4 \) $\max\{\text{PFMCS}[p], x + \sum(A) - \text{PFSF}[p]\}$, term 3 being included iff $\ell > 1$, and term 4 being included iff $p > \alpha_k$.

### 8.2 Handling a Nonnegative $x$ and a Sequence of Type 3

When $A$ is of type 3, we know from Observation 3 that, for every interval $I_i$ and $j \in [\alpha_i .. \beta_i]$, $\text{OMCS}[j] = \beta_i + \lfloor n \rfloor$. Thus, inserting $x \geq 0$ into interval $I_k$ increases the maximal circular sums at the elements of every interval $I_k \neq I_k$ of $A$ by exactly $x$. Moreover, a careful analysis shows that, despite the differences in the interval partition, the maximal circular sums at the other elements of $A^{(p)}$ can be computed exactly as when $A$ is of type 2, which leads to the following result:

**Lemma 7.** If $A$ is of type 3 and $x \geq 0$, then $\text{MCS}(A^{(p)})$ is the maximum among \(1 \) $x + \max\{0, \text{MCS}[p - [n]]\}$, \(2 \) $x + \text{SFMCS}[p]$, \(3 \) $x + \text{IOMCS}[k]$, and \(4 \) $\max\{\text{PFMCS}[p], x + \sum(A) - \text{PFSF}[p]\}$, term 3 being included iff $\ell > 1$, and term 4 being included iff $p > \alpha_k$.

**Proof.** For brevity, we omit the proof. It is similar to the one for type 2 sequences, except for the third item, which was justified above.

### 8.3 Handling a Negative $x$ and a Sequence of Type 2

It immediately follows from Observation 2 that, if $A$ is of type 2 and $x < 0$, then $\text{MCS}_{A^{(p)}}(i) = \text{MS}_{A^{(p)}}(i)$ for all $i \in [0 .. n]$, which implies that $\text{MCS}(A^{(p)}) = \text{MS}(A^{(p)})$. Now, since the preprocessing algorithm for the circular case also computes, when $A$ is of type 2, the arrays used to handle negative values of $x$ in the noncircular case, then we can compute $\text{MS}(A^{(p)})$, and thus $\text{MCS}(A^{(p)})$, in $O(1)$ time exactly as shown in Section 4.

### 8.4 Handling a Negative $x$ and a Sequence of Type 3

Recall that, if $A$ is of type 3, then $\text{OMCS}[i] = \beta_i + \lfloor n \rfloor$ for every interval $I_i$ and $i \in [\alpha_i .. \beta_i]$. Thus, as depicted in Fig. 4, if $p \neq \alpha_k$, then the insertion of $x < 0$ into $I_k$ does not affect the maximal circular sum at any $A[i]$ such that $i \in [\alpha_k .. \beta_k]$, that is, $\text{MCS}_{A^{(p)}}(i) = \text{MCS}(A(i))$. The same is true of any $A[i]$ such that $i \in [\alpha_k' .. \beta_k']$, where $k' = k - \lfloor 1 \rfloor$, if $p = \alpha_k$. It follows that

\[
\max\{\text{MCS}_{A^{(p)}}(i) : i \in [\alpha_k .. p - 1]\} = \text{PFMCS}[p], \quad \text{if } p \neq \alpha_k;
\]

\[
\max\{\text{MCS}_{A^{(p)}}(i) : i \in [\alpha_k' .. \beta_k']\} = \text{IMCS}[k'], \quad \text{if } p = \alpha_k.
\]

Now suppose that $p \neq \alpha_k$ and let $i \in [0 .. n - 1] \setminus [\alpha_k .. p - 1]$: as depicted in Fig. 4, the insertion of $x$ potentially changes the maximal circular sum at $A[i]$, because $x$ is inserted (circularly) between $A(\text{OMCS}[i])$ and $A(i)$. In this case, independently of whether $i \in [p .. \beta_k]$ or not, $\text{MCS}_{A^{(p)}}(i)$ equals either $\text{sum}(A^{(p)}[\text{OMCS}[i] : i]) = \text{MCS}(A(i)) + x$ – which, from the definition of $\text{OMCS}[i]$, is the greatest sum of a subsequence of $A^{(p)}$ which ends in $A[i]$ and includes $x$ – or the greatest sum of a suffix of $A^{(p)}[p : i] = A[p : i] - x$ – which is the greatest sum of a subsequence of $A^{(p)}$ which ends in $A[i]$ and does not include $x$. It then follows that

\[
\max\{0, \max\{\text{MCS}_{A^{(p)}}(i) : i \notin [\alpha_k .. p - 1]\}\} = \max\{x + \text{PFRRMCS}[p], \text{PFRRM}[p]\}.
\]

The zero in the left term of the equation above is a technical detail: $\text{PFRRMS}[p]$ equals the maximum between (1) the greatest sum of a suffix of $A[p : i]$, for any $i$ in the range in question, and (2) zero.

The analysis for the case where $p = \alpha_k$ and $i \notin [\alpha_k' .. \beta_k']$ is similar to the previous one: $\text{MCS}_{A^{(p)}}(i)$ equals either (1) $\text{MCS}(A(i)) + x$ or (2) the greatest sum of a suffix of $A[p : i]$; moreover, the latter term actually equals $\text{sum}(A[p : i])$, since, by Observation 3, every prefix of $A[\alpha_k' : i]$ has nonnegative sum. It then follows that, if $\ell > 1$, then

\[
\max\{\text{MCS}_{A^{(p)}}(i) : i \notin [\alpha_k' .. \beta_k']\} = \max\{x + \text{IOMCS}[k'], \text{IRCS}[k']\}.
\]
Theorem 3. The maximum among IMCS[\(k]\], \(x + \text{IOMCS}[\(k]\) and IRCS[\(k]\], where \(k = |\ell| - 1\), if \(p = \alpha_k\).

(Include the two last terms iff \(\ell > 1\).)

Gathering the results of this section, we finally get:

Theorem 3. Provided that the \(O(n)\) time preprocessing algorithm of Section 7 has already been executed on \(A\), our query-answering algorithm computes \(\text{MCS}(A^{(x-p)})\) in \(O(1)\) worst-case time for any \(x \in \mathbb{R}\) and \(p \in [0..n]\).

9 Concluding Remarks

In this paper we have considered the problem of, for a fixed sequence \(A\) of \(n\) real numbers, answering queries which ask the value of \(\text{MCS}(A^{(x-p)})\) or \(\text{MCS}(A^{(x-p)})\) for given \(x \in \mathbb{R}\) and \(p \in [0..n]\). We showed that, after an \(O(n)\) time preprocessing step has been carried out on \(A\), both kinds of queries can be answered in constant worst-case time. This is both an optimal solution to the problem and a considerable improvement over the naive strategy of answering such queries by means of Kadane’s algorithm (or a variation of it, in the circular case), which takes \(\Theta(n)\) time per query. We showed in Subsection 1.2 that, in the context of finding heuristic solutions to an NP-hard problem of buffer minimization in wireless mesh networks, this improvement reduces the time complexity of a certain column-moving operation on \(m \times n\) matrices from \(\Theta(m \cdot n^2)\) to \(\Theta(m \cdot n)\). Given the generality of these kinds of queries and the multiplicity of applications of the maximal sum subsequence concept, we would not be surprised to see other applications of our algorithms in the future.

The core of the column-moving operation mentioned above is actually an insertion operation: given an \(m \times n\) matrix \(A\) and a size-\(m\) column \(C\), find an index \(p \in [0..n]\) which minimizes \(\text{cost}(A^{(C-p)})\) and return matrix \(A^{(C-p)}\) — recall that \(\text{cost}(A) = \sum_{i=0}^{n-1} \text{MCS}((A[i,0],\ldots,A[i,n-1]))\). An interesting related problem is then that of, given an \(m \times n\) matrix \(A\), inserting \(k\) size-\(m\) columns \(C_0,\ldots,C_{k-1}\) successively and cumulatively into \(A\) according to the criterion in question. By using the algorithms presented in this paper to insert one column at a time, one can carry out \(k\) successive insertions in
\(\Theta(m(n + 1) + m(n + 2) + \ldots + m(n + k)) = \Theta(k(mn + mk))\) time. However, since the input to the problem has size \(\Theta(mn + mk)\), we leave it as an open problem whether substantially more efficient algorithms exist.

**References**

[1] I. F. Akyildiz, X. Wang, and W. Wang. Wireless mesh networks: a survey. *Computer Networks*, 47(4):445–487, 2005.

[2] R. Klasing, N. Morales, and S. Pérennes. On the complexity of bandwidth allocation in radio networks. *Theoretical Computer Science*, 406(3):225–239, 2008.

[3] F. R. J. Vieira, J. F. de Rezende, V. C. Barbosa, and S. Fdida. Scheduling links for heavy traffic on interfering routes in wireless mesh networks. *Computer Networks*, 56(5):1584–1598, 2012.

[4] L. B. Le, E. Modiano, and N. B. Shroff. Optimal control of wireless networks with finite buffers. *IEEE/ACM Transactions on Networking*, 20(4):1316–1329, 2012.

[5] R. C. Corrêa, P. M. S. Farias, and C. P. de Souza. Insertion and sorting in a sequence of numbers minimizing the maximum sum of a contiguous subsequence. *Journal of Discrete Algorithms*, 21:1–10, 2013.

[6] J. Bentley. Programming pearls: algorithm design techniques. *Communications of the ACM*, 27(9):865–873, 1984.

[7] S. An, P. Peursum, W. Liu, and S. Venkatesh. Efficient algorithms for subwindow search in object detection and localization. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 264–271, 2009.

[8] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining with optimized two-dimensional association rules. *ACM Transactions on Database Systems*, 26(2):179–213, 2001.

[9] W. L. Ruzzo and M. Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 234–241, 1999.

[10] M. Csűrös. Maximum-scoring segment sets. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(4):139–150, 2004.

[11] K. Fukuda and T. Takaoka. Analysis of air pollution (PM\textsubscript{10}) and respiratory morbidity rate using K-maximum sub-array (2-D) algorithm. In *Proceedings of the 2007 ACM symposium on Applied computing, SAC ’07*, pages 153–157, 2007.

[12] W. Lu. Improved SWOT approach for conducting strategic planning in the construction industry. *Journal of Construction Engineering and Management*, 136(12):1317–1328, 2010.

[13] D. Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2(3):207–214, 1982.

[14] J. Bentley. Programming pearls: perspective on performance. *Communications of the ACM*, 27(11):1087–1092, 1984.

[15] H.-K. Dai and H.-C. Su. A parallel algorithm for finding all successive minimal maximum subsequences. In J. R. Correa, A. Hevia, and M. Kiwi, editors, *LATIN 2006: Theoretical Informatics*, volume 3887 of *Lecture Notes in Computer Science*, pages 337–348, 2006.

[16] C. E. R. Alves, E. N. Cáceres, and S. W. Song. A BSP/CGM algorithm for finding all maximal contiguous subsequences of a sequence of numbers. In W. E. Nagel, W. V. Walter, and W. Lehner, editors, *Euro-Par 2006 Parallel Processing*, volume 4128 of *Lecture Notes in Computer Science*, pages 831–840, 2006.
[17] G. S. Brodal and A. G. Jørgensen. A linear time algorithm for the $k$ maximal sums problem. In L. Kučera and A. Kučera, editors, *Mathematical Foundations of Computer Science 2007*, volume 4708 of *Lecture Notes in Computer Science*, pages 442–453. 2007.

[18] G. S. Brodal and A. G. Jørgensen. Selecting sums in arrays. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Algorithms and Computation*, volume 5369 of *Lecture Notes in Computer Science*, pages 100–111. 2008.

[19] A. Morihata. Calculational developments of new parallel algorithms for size-constrained maximum-sum segment problems. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming*, volume 7294 of *Lecture Notes in Computer Science*, pages 213–227. 2012.

[20] T. Li-Hui. Sequencing to minimize the maximum renewal cumulative cost. *Operations Research Letters*, 12(2):117–124, 1992.

[21] P. M. S. Farias and R. C. Corrêa. Linear time computation of the maximal sums of insertions into all positions of a sequence. *Electronic Notes in Discrete Mathematics*. Proceedings of the VII Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS 2013). To appear.